

DETECTION AND ANALYSIS OF MEMORY ANOMALIES IN MANAGED LANGUAGES USING TRACE-BASED MEMORY MONITORING

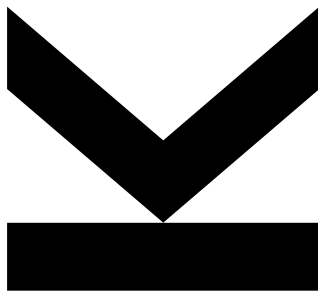
Author
**Dipl.-Ing. Markus
Weninger, BSc**

Submitted at
**Institute for System
Software**

Supervisor and
1st Evaluator
**o.Univ.-Prof. Dr. Dr.h.c.
Hanspeter Mössenböck**

2nd Evaluator
Dr.-Ing. André van Hoorn

June 2021



Doctoral Thesis
to confer the academic degree of
Doktor der technischen Wissenschaften
in the Doctoral Program
Technische Wissenschaften

Sworn Declaration

I hereby declare under oath that the submitted thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.



Markus Weninger
Linz, 15.06.2021

Abstract

Even though modern programming languages such as Java employ automatic garbage collection to free programmers from the error-prone task of manual memory management, anomalies such as memory leaks can still occur. Such anomalies can dramatically impact an application’s performance and can even lead to crashes. Thus, smart tool support is essential to help developers in understanding the memory behavior of complex software systems.

Despite this, most state-of-the-art memory monitoring tools rely on rather limited heap dumps, i.e., they inspect the heap only at a few single points in time. While such approaches may reveal obvious issues, they often do not provide enough details to drill down to the root cause of more complex problems. To tackle this limitation, we propose the use of *memory traces*, i.e., continuous recordings of memory events such as object allocations or garbage collection operations. Such memory traces enable us to reconstruct detailed information about the monitored application’s *memory evolution over time*.

Existing works mainly focus on the (efficient) collection of information-rich memory traces and mostly collect traces for very specific use cases. They often miss discussions of the “big picture”, i.e., the flexibility of memory traces and how they can be used for various kinds of memory analyses.

This thesis revolves around the question how general-purpose memory traces can be processed and leveraged in memory monitoring tools to improve the (semi-automatic) detection and analysis of memory anomalies. It covers data structures and algorithms for memory trace processing, novel anomaly analysis approaches such as automatic data structure growth analysis, as well as interactive visualization techniques. Furthermore, the thesis investigates how (novice) users approach the task of memory analysis and how memory monitoring tools can be improved to better support and guide these users.

All ideas presented in this thesis have been implemented in the memory analysis tool *AntTracks* to showcase their feasibility and applicability.

Kurzfassung

Moderne Programmiersprachen nutzen automatische Speicherbereinigung, um fehleranfällige manuelle Speicherverwaltung zu vermeiden. Dennoch können Anomalien wie Speicherlecks auftreten, die sich drastisch auf die Leistung einer Anwendung auswirken und sogar Abstürze herbeiführen können. Es ist daher essentiell, Entwicklern intelligente Werkzeuge an die Hand zu geben, um das Speicherverhalten komplexer Systemen zu untersuchen.

Die meisten modernen Speicherwerkzeuge nutzen nur Speicherauszüge, d.h. sie inspizieren den Speicher nur zu wenigen bestimmten Zeitpunkten. Solche Ansätze können zwar oberflächliche Probleme aufdecken, bieten aber oft nicht genug Details, um zur Ursache des Problems vorzudringen. Deshalb empfehlen wir die Verwendung von Speicheraufzeichnungen: kontinuierliche Aufnahmen von Speicherereignissen wie Objektallokationen oder Speicherbereinigungsoperationen. Diese erlauben es, detaillierte Information über den *Verlauf der Speicherentwicklung* zu rekonstruieren.

Bestehende Arbeiten fokussieren meist die (effiziente) Sammlung informationsreicher, analysespezifischer Speicheraufzeichnungen, verabsäumen aber oft die Diskussion des „großen Ganzen“: die Flexibilität solcher Aufzeichnungen und ihre Nutzung für verschiedenste Arten von Speicheranalysen.

Diese Arbeit zeigt, wie Speicheraufzeichnungen verarbeitet und genutzt werden können, um die (automatische) Problemerkennung und Speicheranalyse zu verbessern. Sie schlägt Datenstrukturen und Algorithmen zur Aufzeichnungsverarbeitung vor und führt neuartige Anomalieanalysen (z.B. die automatisierte Analyse des Wachstums von Datenstrukturen) sowie interaktive Visualisierungstechniken ein. Ferner untersucht sie, wie (unerfahrene) Benutzer sich bei der Speicheranalyse verhalten und wie Werkzeuge verbessert werden können, um diese Nutzer besser zu unterstützen und anzuleiten.

Alle vorgestellten Ideen wurden im Speicheranalysewerkzeug *AntTracks* umgesetzt, um ihre Machbarkeit und Anwendbarkeit zu präsentieren.

Acknowledgements

This thesis would not have been possible without the help and regular support of many people. First and foremost, I want to thank *Hanspeter Mössenböck*, for being an excellent supervisor and mentor, for his constant feedback and support, and for allowing me to freely follow my research interests. I also want to thank my second examiner *André van Hoorn* for the time and effort he put into evaluating this work and providing feedback.

My sincere gratitude also goes to *Philipp Lengauer* who introduced me to scientific work. He also taught me the basics of garbage collection and memory analysis, and he laid the groundwork for this thesis by helping me to collect experience in the domain of trace-based memory monitoring.

I also want to thank my fellow PhD students, first of all *Andreas Schörghumer*. We were always there for each other when we needed discussions or feedback and never failed to cheer each other up with jokes and memes when we needed a little laugh.

A lot of work has been put into our publications as well as the development of the AntTracks Analyzer and other tools. This work was greatly supported by my student assistants and co-authors *Elias Gander* and *Lukas Makor*. Thank you for your hard work and our countless brainstorming sessions. I also thank all students that I gladly got to (co-)supervise during their bachelor theses, master theses and master projects.

All my love goes out to my long-term partner *Birgit König*, who was always there for me when I needed a shoulder to lean on, a bright mind to talk to, or just somebody that helped me to breathe through and keep track. Thank you for all the understanding you showed when I had little time because of “yet another paper deadline” and all the understanding you showed when I was unnecessarily grumpy because “some bug in the program just took hours to fix”.

A deep hug to my parents, who taught me the important balance between hard work and recreation, the value of friendship and empathy, and who motivated me to chase my goals. They always valued education and supported me in every possible way to help me to deepen my broad interests.

Thanks to my siblings and close friends (you know who you are), who formed me, grew with me, let me grow with them and made me the person that I am today.

This thesis has been developed as part of the *Monitoring and Evaluation of Very-Large-Scale Software Systems (MEVSS)* Christian Doppler Laboratory in cooperation with our industry partner *Dynatrace*. I thank all the people who were involved in this laboratory, especially *Paul Grünbacher*, head of the laboratory, but also a mentor and a great advisor.

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

Contents

Sworn Declaration	i
Abstract	iii
Kurzfassung	v
Acknowledgements	vii
I Introduction and Overview	1
1 Introduction	3
1.1 Outline	3
1.2 Motivation: Memory Anomalies in Managed Languages	4
1.3 Background and Related Work	5
1.3.1 Data Collection	6
1.3.2 Memory Leak Analysis	9
1.3.3 Memory Churn Analysis	12
1.3.4 Memory Bloat Analysis	13
1.4 Remaining Challenges	13
1.5 Contributions	15
1.5.1 Scientific Contributions	15
1.5.2 Technical Contributions	18
1.5.3 Publications	20
2 Overview	23
2.1 Memory Traces and Their Processing	23
2.1.1 Heap Object Classification and Multi-Level Grouping	24
2.1.2 GC Roots and Closures	27

2.2	Data Structure Analysis	29
2.3	Visualization	31
2.3.1	Drill-down Trend Visualization	31
2.3.2	Memory Cities	34
2.3.3	Tree Visualizations	37
2.4	User Guidance and User Behavior	40
2.4.1	Automatic Detection of Suspicious Time Windows	41
2.4.2	Cognitive Walkthrough and User Study	42
2.4.3	Guided Exploration	45
2.5	Memory Churn	47
II Publications		51
3 Memory Traces and Their Processing		53
3.1	Heap Object Classification and Multi-Level Grouping	53
3.2	GC Roots and Closures	66
4 Data Structure Analysis		81
5 Visualization		97
5.1	Drill-down Trend Visualization	97
5.2	Memory Cities	102
5.3	Tree Visualizations	118
6 User Guidance and User Behavior		135
6.1	Automatic Detection of Suspicious Time Windows	135
6.2	Cognitive Walkthrough and User Study	146
6.3	Guided Exploration	184
7 Memory Churn		219
III Future Work and Conclusions		223
8 Limitations and Future Work		225
8.1	Memory Anomaly Evaluation Suite	225
8.2	Automatic Data Structure Detection	226
8.3	Lifetime Analysis	227

8.4	Metric-based Analysis	228
8.5	Visualization Extensions	228
8.6	Using Visualizations in SE Education	229
8.7	Static and Dynamic Analysis Synergies	229
8.8	IDE Integration	230
9	Conclusions	231
	Appendices	235
A	Memory Cities Artifact	237
	Bibliography	243
	Curriculum Vitae	263

Part I

Introduction and Overview

Chapter 1

Introduction

1.1 Outline

This thesis is designed as a cumulative dissertation. It contains all publications that have been published as part of my PhD research, explains their core concepts as well as the motivation behind them, and puts the work into perspective to the state of the art and related work. Thus, this thesis is structured into three parts:

- Part I establishes the context of this dissertation. It presents the motivations that led to our research (Section 1.2), outlines background and related work in memory monitoring (Section 1.3), discusses challenges in this field that have been tackled in this thesis (Section 1.4), and gives an overview of our scientific and technical contributions (Section 1.5). Section 2.1 through Section 2.5 summarize the core ideas of the published papers, separated into thematic categories, and illustrate how these topics are related to each other.
- Part II contains the published papers, grouped into the above mentioned thematic categories.
- Part III concludes the thesis by outlining possible future work and summarizing final conclusions.

This thesis contains some collective references for certain terms. This is done to achieve a combined bibliography at the end of the thesis that contains *all* references that have been used in at least one of the published papers.

Nevertheless, each paper in Part II is self-contained and thus again has its own bibliography with more details provided for the cited works.

1.2 Motivation: Memory Anomalies in Managed Languages

In traditional *unmanaged programming languages* such as C, it is the programmer's responsibility to correctly handle memory allocations and deallocations. While this offers a high degree of flexibility, which makes it possible to write highly-optimized code, manual memory management can also easily lead to unintended memory bugs and errors. This problem is probably best summarized with the well-known mantra "*with great power comes great responsibility*".

Typical memory-related problems in unmanaged languages encompass problems such as memory leaks or dangling pointers. Memory leaks occur if the programmer forgets to free memory once it is not needed anymore (for example, by missing a needed `free()` call), unnecessarily increasing the applications memory footprint over time. A dangling pointer access occurs if the developer tries to access memory that has already been freed, e.g., if a `free()` call has been placed too early in the code. Thus, this is also called a *use after free* problem.

To reduce the number of memory-related problems, modern high-level programming languages such as Java employ a *garbage collector* (GC) to automatically reclaim unused memory. Such *managed languages* keep track of all static fields and thread-local variables (so-called *GC roots*¹) as well as all the references between heap objects. During a garbage collection phase, objects that are no longer (indirectly) reachable from these GC roots, i.e., objects that can no longer be referenced from code, are then automatically reclaimed by the GC, freeing up their reserved memory. This approach relieves the programmer from the error-prone task of manual memory management, which also makes certain memory problems of unmanaged language such as dangling pointer accesses impossible. Nevertheless, garbage collection comes with its own set of possible memory anomalies that can slow down applications if developers handle object allocations and object storage carelessly.

¹In Java, other GC root types such as *JNI* or *Monitor* also exist [317].

One of the main memory anomalies that can occur in managed languages is a *memory leak*, although memory leaks in managed languages differ from those in unmanaged languages. As explained before, in unmanaged languages, a memory leak occurs when the programmer forgets to include a *free* call at the correct location. In managed languages, we can define a memory leak as a situation in which objects that are no longer needed remain reachable from GC roots due to programming errors [192]. For example, a developer may forget to remove objects from long-living data structures once they are not needed anymore. These objects remain reachable, thus cannot be reclaimed by the garbage collector, and will therefore accumulate over time. Besides memory leaks, other kinds of memory anomalies exist. *Memory churn*, also called *excessive dynamic allocations* [227, 270] or *high allocation density* [75], occurs when objects are (unnecessarily) allocated in high frequencies, just to be reclaimed shortly after their creation. *Memory bloat* [135, 199, 349] describes the problem of using memory in an inefficient way, causing a high memory overhead to achieve relatively simple tasks. It is often caused by heavily using (object-oriented) abstractions such as over-generalized data structures. Different *patterns of memory inefficiency* [56, 57] can also be regarded as memory anomalies, i.e., patterns that lead to a sub-optimal use of memory. Examples of such patterns are unnecessarily using empty arrays and lists instead of `null` to express empty state, or to use lists instead of arrays for fixed-size data structures.

Memory anomalies can dramatically impact an application’s memory utilization and garbage collection behavior, and thus its performance. Anomalies such as memory leaks can even lead to out-of-memory crashes. Thus, tool support including smart (semi-automatic) analyses and easy-to-use features is essential to help software engineers in finding and fixing memory anomalies as well as understanding and improving the memory behavior of complex software systems.

1.3 Background and Related Work

In this section, we give an overview of the basics of memory monitoring and explore related work in the domain of memory analysis. We mainly focus on approaches for the programming language Java and its underlying Java virtual machine (JVM), but the ideas behind these approaches can also be

transferred to other managed languages and runtimes such as C# and its .NET Common Language Runtime (CLR).

First, we present various kinds of data collection techniques, mainly comparing *heap dumps* to *memory traces*. Next, we discuss different kinds of memory anomalies such as memory leaks, high memory churn, and memory bloat. These memory anomalies cause the monitored application to exhibit certain memory patterns. We give an overview of related works, how they detect these patterns and how they make use of them.

1.3.1 Data Collection

Before one can investigate an application's memory behavior, information about its internal operation has to be collected. If this collection is performed without executing the application, e.g., by analyzing the source code, we speak of *static analysis*. On the other hand, *dynamic analysis* approaches collect their data while the application is running. *Dynamic analysis* approaches can be separated into *online* and *offline* approaches; a similar taxonomy is to group them into *synchronous* and *asynchronous* monitoring approaches [48]. Online approaches directly inspect and react to the monitored application while it is running, for example by checking at run time whether certain violations occurred or conditions hold. Offline approaches, on the other hand, collect data at either a *single point in time* (e.g., a heap dump), *multiple points in time* (e.g., multiple heap dumps), or *continuously* (e.g., a memory trace) for later analysis in a separate *offline analysis tool*.

Static Analysis Static analysis does not rely on recordings performed during program execution, but rather inspects the source code (or other static artifacts) without taking run-time information into account. For example, the static analysis tool LeakChaser [353] is based on the observation that objects outside of a loop often keep unnecessary references to objects created inside a loop. It uses static analysis to identify such unnecessary references and reports them to the user. Distefano and Filipović [72] present a static analysis approach that uses shape analysis to identify certain objects that are reachable but no longer used. The static approach by Shaham et al. [267] can detect no-longer used regions of arrays, allowing the garbage collector (that has been modified to know about these regions) to collect them earlier. A regular problem of static memory analysis approaches is that they often do not scale well for real-world applications due to their complexity.

Type Histograms Most dynamic memory analysis tools can at least display a type histogram [213], i.e., they query the running program to report how many objects and bytes of each type exist at the current point in time. In Java, it is even possible to obtain a type histogram without using an external tool and without writing a heap dump to disk by setting the `-XX:+PrintClassHistogramBeforeFullGC` and the `-XX:+PrintClassHistogramAfterFullGC` flag [172]. These flags cause a textual class histogram to be printed to the console before and after every full garbage collection. Even though type histograms are easy to obtain, they only provide superficial, aggregated information. Thus, most memory analysis tools resort to either full heap dumps or to even more flexible memory traces to perform their analyses.

Heap Dumps Most state-of-the-art memory monitoring tools rely on information reconstructed from a single heap dump (or possibly on information from comparing two heap dumps), which can be limiting in many cases. The data stored in a heap dump can be used to reconstruct the state of the heap at one specific point in time, i.e., to reconstruct information about the objects that were live at this point in time, as well as their references to each other. To create such a heap dump, tools shipped alongside Java such as HPROF [215, 223] or jmap [213] can be used. Popular openly available tools that use heap dumps as their data source are *VisualVM* [221], the Eclipse Memory Analyzer (MAT) [79], or *jhat* [218]. Popular closed source tools to analyze heap dumps are JProfiler [80] or YourKit [354].

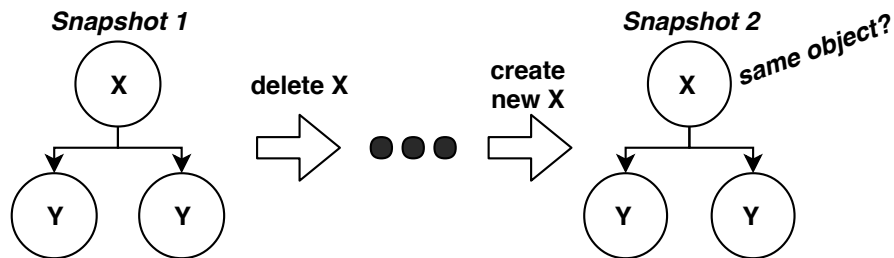


Figure 1: Analyses based on heap dumps lack information on *object identity*.

While approaches and tools that rely on heap dumps may reveal obvious issues, they generally do not provide enough details to drill into more complex problems and their root causes. Most notably, they do not provide any

information on how the memory *evolves over time*. While it is possible to use multiple heap dumps to inspect the *aggregated* heap growth (e.g., the increase of objects of a certain type), heap dumps do not have a concept of *object identity*. Using only heap dumps without additional information, it is not possible to decide whether an object that was alive in a certain dump is still alive in a later dump. Figure 1 illustrates this problem, where we can see two heap dumps (also called *snapshots*). We can only infer that both snapshots contain one X object, but not whether these two X objects are actually the same.

Another disadvantage is that writing heap dumps is time-consuming and halts the application while dumps are produced. Thus, it is often not feasible to record a heap dump (not to speak of regular or frequent heap dumps) in applications that must be responsive. For example, it would probably not be possible to record a heap dump in a large server application that has to continuously answer web requests without disrupting users. Heap dumps are thus often recorded as a “last resort” when an `OutOfMemoryError` occurs, to provide the developer at least with a single heap snapshot of when the application crashed.

Continuous Monitoring and Memory Traces Continuous recordings of the memory behavior of applications are called *memory traces*. They provide data to perform detailed analyses based on the evolution of an application’s heap. These analyses include growth analyses, staleness measurements, or visual analyses, and will be discussed later in more detail.

Typical memory traces [55, 117, 118, 130, 231, 238, 239, 247, 248, 347, 355] encode events such as object allocations, garbage collection operations, object deaths, or object field accesses. The granularity and the level of detail may vary significantly between different tracing approaches, since most approaches produce trace formats for specific kinds of analyses. There are also other types of traces such as *execution traces* [63, 64, 134, 290] that rather focus on call hierarchy information instead of the contents of the heap.

To generate traces, typically one of the following techniques is used [22]: (1) A *modified execution environment* such as a custom Java VM that can access internal information; (2) a *sampling-based approach*, e.g., an agent using the Java VM Tool Interface [211] to receive periodical callbacks about memory-relevant events in the application; or (3) an *instrumentation-based approach* that relies on adding code to an existing application, either before

compilation (for example, by using aspect-oriented programming [153] with AspectJ [51, 151, 152]) or at run time (for example, by using a bytecode modifying library such as ASM [42, 43, 165] or Javassist [52–54]).

A major downside of typical memory tracing approaches so far was that their recording often caused a several 100-fold slowdown of the analyzed application [346], making them unfeasible for use in production systems. Also, since modifications of real-world virtual machines such as the Hotspot VM [219] require a deep understanding of its internals, many approaches have only been implemented in research VMs such as the Jikes VM [137] or the Maxine VM [342] that are more easily approachable, but are (as their name suggests) hardly ever used outside of academia. Yet, Lengauer et al. [173–175] have recently shown that it is possible to generate general-purpose memory traces in a real-world virtual machine, introducing only a few percent of run-time overhead. Thus, we hope to see more implementations of memory tracing techniques in production VMs in the future that provide a detailed data basis for interesting analyses.

1.3.2 Memory Leak Analysis

Memory leaks are probably the most intensively researched memory anomalies, since they are also the most often reported memory defects [100, 101]. As already explained, memory leaks in managed languages occur if objects that are no longer needed remain reachable from garbage collection roots (e.g., static fields or local variables) due to programming errors. Such leaks lead to a growing memory footprint, which at some point will cause an application to crash. In the following, we will briefly discuss different kinds of memory analysis approaches, following a similar memory leak analysis taxonomy as Sor and Srirama [278].

Detecting *Staleness* [33–35, 114, 238, 266, 289, 350, 352] The staleness of a heap object is expressed by the time since the program last actively used it, i.e., the last time someone wrote to it or read from it. Staleness analysis approaches assume that live objects which are not used for a long time are more likely to be involved in a memory leak. However, the proposed techniques are hardly used outside of academic research due to the extremely high costs involved in tracking and recording accesses to objects.

One exemplary staleness analysis approach that addresses this scalability issue is *container profiling* by Xu and Rountev [350, 352]. In their approach, they do not track accesses to single objects, but rather focus on accesses to data structures. For each data structure, they track operations such as **ADD**, **GET**, and **REMOVE**. As these operations may look different for various data structures (e.g., an **ADD** operation on a list may be called **add** while it may be called **put** on a map), their approach requires ahead-of-time modeling of container wrappers. The developer has to introduce a “glue layer” in the monitored application’s source code that maps methods of each container type to the mentioned primitive operations via annotations. This way, their approach can track when data structures last interacted with their containing objects. An important difference between this approach and other more traditional staleness analysis approaches is that it calculates staleness based on the time since an object was last placed into or retrieved from a collection, rather than the time since the object was generally accessed. This is due to the authors’ assumption that most memory leaks involve growing and wrongly-used data structures.

Detecting Growth [51, 144–146, 200, 275–277] Approaches that detect growth typically inspect the object reference graph, i.e., the references between the heap objects, and analyze how this graph changes over time to detect growing patterns. For example, Jump and McKinley [144–146] implemented Cork, an online tool integrated into the Jikes RVM, that constructs a *Type Points-From Graph (TPFG)* during each garbage collection. A TPFG contains a node for each type and the directed edges between the nodes summarize how many objects of a given type are referenced by objects of the type on the edge’s other end. As the application runs, multiple TPFGs are created and stored, introducing about 1% memory overhead and 2.3% runtime overhead. Growing parts in these graphs, i.e., growing types of data structures, are then reported to the user. This directs the developer’s attention to certain data structures that exhibit continuous growth. There are also growth analysis approaches that involve statistics and machine learning [275–277], and many growth analyses also involve visualizations, as we will discuss in the next paragraph.

Visualization Techniques [2, 70, 119, 120, 150, 198, 226, 229, 241–243, 255, 360] Most memory visualizations revolve around object refer-

ence graph visualization. A pure object graph consists of nodes that represent heap objects and edges that represent the references between them [228]. Even though such a graph could be directly visualized as a node-link diagram [360], the size of real-world applications (having millions of live objects) renders approaches that display every heap object as a separate node infeasible. Thus, most approaches create *ownership trees* using the concept of *object ownership* [18, 197, 228, 317] based on the *dominator relation* [177]. These concepts will be explained in more detail in Section 2.1.2. In general, ownership trees can be used to detect single objects that keep many other objects alive and are often used as a basis for further graph aggregations.

Reiss [241–243] visualizes an aggregated ownership graph in an icicle-like visualization [11, 115, 164] using coloring, hatching, hue and saturation to convey information about the underlying growth. The approach by Hill et al. [119, 120] plots the evolution of ownership trees in a scalable tree visualization that shares visual similarities with flame graphs [109, 110]. Mitchell et al. [198] apply further transformations on ownership trees to detect costly data structures, which are then displayed in a node-link diagram. Heapviz [2, 150] is a tool that also focuses on data structures. It allows users to display them on different levels of detail, arranging collapsible nodes in a radial node-link diagram. The work by De Pauw and Sevitsky [70, 226] is one of the few object graph visualization approaches that does not utilize the dominator relation. Instead, they extract reference patterns, i.e., repetitive reference sequences in the heap object graph, and then visualize these reoccurring patterns. The detection of these patterns can be restricted to those objects that have accumulated between two heap snapshots, i.e., potentially leaking objects.

Even though various work regarding memory visualization exists, most state-of-the-art tools do not yet make use of them. We think that this might be the case due to the complexity of the visualizations, still requiring deeper understanding of how managed runtimes work internally. Instead, most tools visualize the growth of the heap just as a line plot or stacked area plot, for example as done in tools such as JConsole [217], VisualVM [221] or Kieker [154, 303, 304]. In these simple visualization approaches, the monitored application’s memory footprint is periodically plotted in a time-series chart. The user can then check these charts for suspicious sections of continuous growth that might hint at a memory leak. If such a time window is found, the user then continues the analysis in traditional (table-based) analysis views.

Static Analysis [72, 267, 353] As already laid out in Section 1.3.1, static analysis approaches do not use data collected at run time, but rather inspect the application’s source code (or other static artifacts).

Hybrid Techniques Hybrid memory leak analysis techniques combine several of the mentioned techniques. For example, based on the work by Xu and Rountev [350, 352] Yu et al. [356] present a memory leaking confidence (MLC) metric, which combines information about *object staleness* and *object growth*. Other examples for hybrid techniques involve static analysis in combination with dynamic analysis [81], or approaches that use visualizations to better communicate growth behavior. Even though some of the approaches referenced in the above paragraphs may be classified as hybrid techniques, we presented them in the paragraph with their strongest focus.

1.3.3 Memory Churn Analysis

Another common memory anomaly that is often overlooked by novice users is high memory churn [75, 227, 270]. High memory churn, i.e., the excessive allocation and deallocation of large amounts of objects, results in run-time overhead that could often be avoided. The overhead stems from the facts that the allocation of an object itself takes time and that the large number of dying objects results in an increased number of garbage collections. A situation that often leads to high memory churn is the allocation of temporary short-living objects in heavily executed loops. In each loop iteration, new objects are allocated that almost immediately turn into garbage. Another typical problem is the use of boxed primitives as generic types [40], e.g., `ArrayList<Integer>`. Every time a primitive is added to such a structure, it is wrapped into a heap object, which causes unnecessary memory overhead.

In summary, the main contributors to high memory churn are objects that are allocated in bursts and do not survive a single garbage collection. However, obtaining the information on how long objects survive is expensive, especially if the time of death should be reconstructed exactly [118, 247, 248]. Thus, most tools do not provide information about object age, but limit the user to inspect the plain number of allocations that happen within a certain time window. Memory churn is often detected visually by spotting frequent spike patterns in memory charts (i.e., heavily growing memory consumption followed by many object deaths) or by plotting the aggregated number of

allocations per minute (i.e., detecting allocation-intensive time windows), as it is done in Dynatrace [77].

1.3.4 Memory Bloat Analysis

Memory bloat [135, 199, 349] describes the inefficient use of memory for achieving relatively simple tasks. It is often caused by heavily using (object-oriented) abstractions, such as in over-generalized data structures. For example, Costa et al. [65] have shown that by choosing alternative data structures instead of the default Java collections can lead to memory savings of up to 88%, depending on the application and use case. Most techniques for memory bloat detection focus on data structure analysis, searching for those structures that require many auxiliary objects [201, 202], or analyze the inefficient use of data structure operations for adding, getting, or removing elements [348, 351].

1.4 Remaining Challenges

As discussed, advanced memory anomaly analyses often utilize memory traces, as they provide continuous information that enables analyses regarding object staleness, growth, and similar other factors. Unfortunately, the majority of related works on memory traces either (1) focus on efficient ways to collect the data without presenting novel analysis techniques, or (2) collect specific memory traces that are used for one specific analysis, but they do not explore if their traces could also be used for other analyses.

Thus, this thesis revolves around the usage of general-purpose memory traces (i.e., traces that do not focus on the analysis of a specific problem) such as the traces produced by the AntTracks VM by Lengauer et al. [173–175]. More specifically, since it has been shown that such traces can be collected efficiently, even in real-world VMs running production systems, the main remaining challenge we want to tackle in this work is to show how they can be utilized in developer-oriented memory analysis tools. To investigate this, we focus on questions and challenges on how to *process and utilize* memory traces, for example by providing a common data basis for *various* analyses and visualizations to help developers in *detecting, analyzing and fixing* memory anomalies.

In summary, open questions in the following areas are tackled throughout the thesis:

- **Memory Trace Processing** Memory traces are textual or binary recordings of memory events. To make sense of these events, the traces have to be processed and brought into a useful format. Memory traces have two main advantages over heap dumps: (1) They can provide vastly more information about every heap object (such as the object’s type, allocation site, its allocating thread, its references to other objects, and much more), as well (2) information about the *evolution* over time, e.g., how long certain objects survive.

This opens a number of interesting questions to investigate, including:

- How can information that is reconstructed from a memory trace be arranged to build a general data basis for a variety of analyses, i.e., is there a flexible yet useful format to represent data reconstructed from a trace?
 - How can we use this aggregated information in combination with information about the references between objects in novel analyses?
 - How can we reconstruct and utilize lifetime information, i.e., the birth and death of objects, in analyses?
- **Data Structure Analysis** Since memory leaks are often caused by growing data structures (lists, trees, hash maps, etc.) [57, 200, 350, 352], how can memory traces be used to automatically detect and inspect those data structures that are most likely involved in a memory leak? To make this question more distinct from previous research, the analysis should not be restricted to *single-object ownership*, i.e., leaks caused by a single data structure. Instead, we also want to explore how we could detect leaks caused by the interaction of multiple data structures, and how to report them to the user.
 - **Visualization** Existing visualizations are often *static*, visualizing the heap at one specific point in time or visualizing some difference-measure between two points in time. Consequentially, most techniques do not focus on the visualization of *general trends* in the monitored application, i.e., they do not focus on the *evolution of the heap memory over*

time. Thus, an open question that is tackled in this thesis is how to provide accessible and understandable visualizations to depict the often complex *continuous* memory evolution reconstructed from memory traces. For example, continuous memory growth may hint at the possible existence of a memory leak, yet only certain types or data structures are responsible for it. Providing (1) *interactive* visualizations that (2) emphasize the memory evolution (and growth) over time and (3) provide a drill-down into suspicious parts of the heap may help to detect and understand the root causes of memory anomalies.

- **User Guidance and User Behavior** Most memory tracing approaches have only been evaluated with regard to their performance overhead, yet the analyses and tools built upon them are nearly never evaluated with users. Thus, this thesis should also investigate how users behave during memory analysis. Since most memory analysis tools are used by “normal software developers”, i.e., users with a limited background in memory monitoring, so-called expert tools may be too complex for this kind of users [222, 288]. We want to investigate how novices can be better supported by memory monitoring tools by guiding them through memory analyses, automatically pointing them to suspicious memory behavior.

1.5 Contributions

The contributions presented in this thesis can be divided into scientific contributions, technical contributions, and their corresponding publications.

1.5.1 Scientific Contributions

The scientific contributions of our approach can be summarized as follows:

Memory Traces and Their Processing [317, 331] We present a novel approach on how to enable a vast amount of different analyses through a flexible heap object classification system. At its core, it allows the user to freely group heap objects according to any of their properties such as type, allocation site, allocating thread, or any user-specified criterion. Grouping objects according to multiple criteria, for example by first grouping all objects

by their types and then all objects of a given type by their different allocation sites, results in a *memory tree*. We have shown that such memory trees are a suitable means for data representation, as they can be used as a source for analyses such as heap state analysis, data structure growth analysis, as well as different visualization techniques. Our work on heap object grouping and memory trees has been nominated for the best paper award at ICPE 2018.

Another topic to which we contributed is the analysis of object ownership, i.e., analyzing which objects keep each other alive. Most existing approaches rely on a *dominator tree* constructed from the references between the heap objects to perform such ownership analyses. Yet, these approaches lack the ability to report problems that are caused by multi-object ownership [191], i.e., where leaking objects are kept alive by more than a single object. We present algorithms to calculate the *transitive closure*, i.e., the set of all objects that are reachable from some other object, as well as the *GC closure*, i.e., the set of all objects that also become eligible for garbage collection if a certain object dies. Further, these closures may not only be computed for a *single* root object (single-object ownership), but also for a set of root objects (multi-object ownership). This allows us to derive metrics such as *deep size* and *retained size* for any group of objects, which integrates well into our heap object grouping mechanism. These metrics provide vital information to detect suspicious memory patterns, even those that involve multi-object ownership.

Data Structure Analysis [316, 318] The previously mentioned closure algorithms are especially useful when analyzing the evolution of data structures over time, since they allow us to detect *growing data structures* and, more specifically, the kind of growth these data structures exhibit. We developed a domain-specific language that can be used to express which parts of a data structure are *internal* (such as the nodes of a `HashMap`) and external (such as the objects stored as keys and values in a `HashMap`). Combining this information with the information about the transitive closure and the GC closure growth of every data structure allows us to derive metrics to rank data structures according to their probability of being involved in a possible memory leak. At the same time, we can provide information about the nature of this leak being either data-structure internal or external, as well as being either due to single-object ownership or multi-object ownership.

Visualization [325–328, 330] We present novel techniques to visualize the evolution of memory trees *over time*. For this, we studied existing visualization approaches and evaluated them with respect to their applicability and usefulness for visualizing memory metrics, especially the evolution of memory trees. We present adaptations and applications of traditional time-series charts, 2D *tree visualizations* as well as an engaging 3D *software city* visualization. All these approaches provide novel inspection and interaction techniques that make the memory evolution of system easier to understand and more tangible. For each approach, we present a complete visualization pipeline [73, 183], including data pre-processing, layouting, and user interaction. Both of our papers presented at STAG 2020 (2D tree visualizations) and VISSOFT 2020 (3D memory cities) have been awarded with the best paper award.

User Guidance and User Behavior [315, 319, 321] We performed a structured cognitive walkthrough as well as a user study to evaluate the usefulness of our techniques as well as to gain understanding on how novice users use memory monitoring tools and their analysis capabilities. Based on the results, we derived a set of recommendations that may guide memory tool developers in improving existing analyses and implementing new features.

Following these recommendations, we also improved the user guidance in our own memory analysis techniques. We devised algorithms that are able to automatically detect patterns in an application’s memory utilization that hint at certain memory anomalies such as memory leaks or high memory churn. If a suspicious time window is detected, it is automatically highlighted, which relieves the users from having to find such patterns themselves. Based on this first experience, we developed a complete user guidance concept called *guided exploration*, which revolves around four support steps that tools should perform: automatic *detection* of suspicious patterns, *highlighting* relevant UI elements, *explanation* why the observed patterns are undesirable and *suggestion* of suitable next steps. The implementation of guided exploration in our AntTracks Analyzer tool provides these support operations on every step during memory leak analysis as well as high memory churn analysis.

Memory Churn Analysis [320] We did not only investigate memory leaks, arguably the most frequent memory anomaly, but also other prob-

lems. For example, we present a technique that automatically hints at memory churn hotspots, i.e., time windows in which a large quantity of objects are allocated and freed shortly afterwards. We further show how we reconstruct the lifetime of objects, i.e., how many garbage collections each object survived, from memory traces and how this information can be incorporated into memory trees for more detailed memory churn analysis.

1.5.2 Technical Contributions

The major technical contribution of this thesis is the development of a set of tools for memory analyses based on memory traces, especially the *AntTracks Analyzer* tool that can process traces produced by the *AntTracks VM*. Further, we developed tools for visualizing an application’s memory evolution over time, most notably the visualization tools *Memory Cities* and *WebTreeViz*.

Many parts of the AntTracks Analyzer have been developed together with Elias Gander, who contributed to its implementation as a student assistant and as part of his master’s thesis [98]. Lukas Makor contributed to the implementation of our visualizations as a student assistant, as part of his bachelor’s thesis and as part of his master’s thesis [188].

AntTracks VM The *AntTracks VM* [173–175] is a virtual machine based on the Java Hotspot VM [219]. It was initially developed by Lengauer et al. and has been extended by us as part of this thesis. The VM records events such as object allocations and object movements during garbage collection by writing them into trace files [173, 174], introducing a run-time overhead of about 5%. To reduce the trace size, the VM does not record any redundant data and applies compression [175]. As part of this thesis, the VM’s tracing mechanism has been extended to also record information about garbage collection roots, a vital information for memory anomaly detection. For example, GC root information allows us to find those static fields and local variables that keep a lot of other objects alive (top-down analysis). GC root information also helps during bottom-up analysis, e.g., by recursively following the incoming references of accumulating objects up to the GC roots that are keeping them alive.

AntTracks Analyzer The AntTracks Analyzer is the core tool that has been developed throughout this thesis. It is written in Java and Kotlin with a JavaFX user interface and serves as a reference implementation of all ideas and concepts presented in this thesis. All presented techniques, starting from heap object classification and multi-level grouping over closure calculation and data structures growth analysis up to memory churn analysis and user guidance, have been implemented in this tool. The tool has been extensively used throughout all publications to present the feasibility, applicability and usefulness of the presented techniques. It is openly available for download [314] and can be further extended in the future.

Memory Cities Inspired by the CodeCity tool by Wettel [334], we developed the *Memory Cities* tool that visualizes the evolution of an application’s heap over time using the *software city* metaphor. While in the past, software cities have mostly been used to visualize static metrics of a software system (such as class hierarchies), we adopted the idea to visualize the dynamic memory behavior of an application. Groups of heap objects (e.g., objects of the same type allocated in a certain method) are visualized as buildings that are arranged in districts (e.g., all buildings concerning the same type). A building’s size corresponds to the number of objects it represents. By continuously updating the city over time we are able to create the engaging feeling of an evolving city, where growing buildings represent growing parts of the heap. Using color and opacity, users are specifically attracted to certain buildings, which can then be interacted with to inspect them in more detail.

The tool has been successfully evaluated as an artifact at the VISSOFT 2020 conference. It is available at [329], a video of it can be found at ², and instructions on how to use the tool can be found in Appendix A.

WebTreeViz This web-based tool can display the heap evolution over time either in a sunburst or an icicle visualization, two widely used tree visualization techniques. It can be used to easily inspect what kind of objects take up the most space in the heap, and how this heap composition changes over time. For this, the tool processes a list of memory trees, i.e., the grouped heap state of the monitored application at different points in time, and provides two interaction techniques to inspect their evolution: In the *time-travel-based*

²<http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.mp4>

visualization, a single space-filling tree visualization shows the monitored application’s heap memory at a given point in time. Users can step back and forth in time, causing the visualization to update itself, similar to our memory city approach. In the *timeline-based* visualization, a time-series chart depicts the overall memory consumption over time. Above this chart, multiple memory tree visualizations are shown side-by-side for a number of user-selected points in time. These tree visualization can then be compared to each other to extract information about changes in the heap. Both visualizations help users to gain new insights and to detect (problematic) memory trends in their applications.

The tool (directly usable within the browser) is available at ³. There, we also provide demo data to test the tool, i.e., the data that has been used in the case studies presented in the respective paper [330]. A video in which we explain the tool’s most important features can be found at ⁴.

Interoperability Although we based Memory Cities and WebTreeViz on data provided by AntTracks, we also defined a *JSON* interface through which the tools can load their data either from the file system or via a *Web-Socket* [85, 332] connection. Using this JSON interface, our visualization tool could also be used with monitoring tools other than AntTracks.

1.5.3 Publications

The results of this thesis were published in various conference proceedings and journals. Table 1 shows all publications (all first-authored) sorted in chronologically ascending order. Part II provides the full text of all publications, grouped by thematic categories. Table 2 lists four more (co-)authored publications that are not central to this dissertation.

³<http://bit.ly/STAG-MemoryTreeVizTool>

⁴<http://bit.ly/STAG-MemoryTreeVizVideo>

Table 1: Chronological list of core publications (all first-authored)

Ref	Title	Conference / Journal
[331]	User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring.	ICPE'18 (Best Paper Nominee)
[317]	Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring.	ManLang'18
[316]	Analyzing the Evolution of Data Structures in Trace-Based Memory Monitoring.	SSP'18
[318]	Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection.	ICPE'19
[325]	AntTracks TrendViz: Configurable Heap Memory Visualization Over Time.	ICPE'19
[319]	Detection of Suspicious Time Windows in Memory Monitoring.	MPLR'19
[327]	Memory Leak Visualization using Evolving Software Cities.	SSP'19
[321]	Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study.	EICS'20 / PACMHCI (Journal)
[326]	Memory Cities: Visualizing Heap Memory Evolution Using The Software City Metaphor.	VISSOFT'20 (Best Paper)
[328]	Heap Evolution Analysis Using Tree Visualizations	SSP'20
[320]	Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance	SSP'20
[330]	Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations	STAG'20 (Best Paper)
[315]	Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools	EICS'21 / PACMHCI (Journal)

Table 2: Chronological list of non-core publications

Ref	Title	Conference
[173]	Efficient Memory Traces with Full Pointer Information. (Co-authored)	PPPJ'16
[176]	A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. (Co-authored)	ICPE'17
[323]	User-centered Offline Analysis of Memory Monitoring Data. (First-authored)	ICPE'17
[322]	Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment. (First-authored)	APSEC'18

Chapter 2

Overview

In this chapter, we give an overview of the topics presented in this thesis. We describe our core ideas and the respective papers and how they relate to each other, dividing them into five main areas: Section 2.1 introduces the reconstruction and aggregation of heap object information from memory traces, Section 2.2 introduces our (semi-automatic) analysis of data structures and their growth behavior, Section 2.3 introduces our visualization techniques to support memory analysis, Section 2.4 discusses our work on usability, user behavior and user guidance, and Section 2.5 introduces our inspection technique for memory churn.

2.1 Memory Traces and Their Processing

Most state-of-the-art memory monitoring tools rely on heap dumps and use a type histogram, i.e., a table showing how many objects of each type were live at the given point in time, as their main way of displaying information. Yet, in addition to such type histograms, users might also want to know where these objects have been allocated, which other objects refer to them, or how long they survived. Luckily, memory traces allow us to collect and reconstruct this information [23].

Still, memory traces on their own are not a universal remedy. The data in them is often reduced to a minimum, which makes post-processing steps necessary to reconstruct information about the various heap objects. Furthermore, since modern object-oriented applications often accommodate millions of live objects at any point in time, inspecting them on the object-level, i.e.,

one-by-one, becomes infeasible and makes aggregation necessary. Therefore, the heap objects first have to be aggregated into a format that is flexible enough to suit different kinds of memory analysis techniques.

In this section, we discuss our aggregation concept of *heap object classification* and *multi-level grouping*. A classifier is a function that groups heap objects according to a certain criterion such as their type, their allocation site, or their allocating thread. Grouping the objects according to multiple classifiers results in a hierarchical *memory tree*. For instance, the top level may group objects by their types, the second level by their allocation sites, and the third level by their allocating threads. The top level gives users a coarse overview of the application’s memory distribution, and drilling down into suspicious objects groups allows users to inspect them in more detail, following Shneiderman’s well-known mantra ‘*Overview First, Zoom and Filter, Details on Demand*’ [269]. Memory trees are a recurring topic in this thesis, as they are the main data structure that we use for all our analyses.

In addition to that, we discuss how we modified the AntTracks VM to not only collect information about heap objects but also about GC roots, e.g., static fields and local variables, that keep objects alive. GC root information is vital for a memory monitoring tool to be able to determine which roots ultimately cause certain objects to accumulate. Useful concepts that we use to describe *keep-alive* relations are *closures* and their respective metrics *deep size* and *retained size*, which allow us to detect objects and object groups that keep a large number of other objects alive.

2.1.1 Heap Object Classification and Multi-Level Grouping

In *User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring* [331], we present our aggregation concept to build *memory trees*. A memory tree is an aggregated representation of a heap state, i.e., of all heap objects that were live at a certain point in time. In it, these heap objects are grouped according to a selected set of *object classifiers* that group objects based on common properties. Both, memory trees and objects classifiers are explained in more detail in the following.

General approach: To build a memory tree, once a heap state is reconstructed from a trace file, its heap objects have to be grouped according to

certain criteria provided by user-definable heap object classifiers. In general, a heap object classifier has a name, a description and a `classify()` function that takes a heap object as its input and returns a classification result. For example, the *type classifier* takes in a heap object and returns this object's type as classification result (e.g., `HashMap`), while the *allocating thread classifier* returns the object's allocating thread as result (e.g., `Input Worker Thread`). Heap objects with the same classification result will then be grouped into the same group.

Grouping the heap objects according to the classification results of multiple classifiers results in a hierarchical *memory tree*. A common classifier combination is to first group all heap objects by their types (using the type classifier) and then by their allocation sites (using the allocation site classifier), as shown in Figure 2. In this figure, yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object[]`, of which the objects 0, 1 and 3 have been allocated in `Stack:init()` and object 2 has been allocated in `MyService:foo()`.

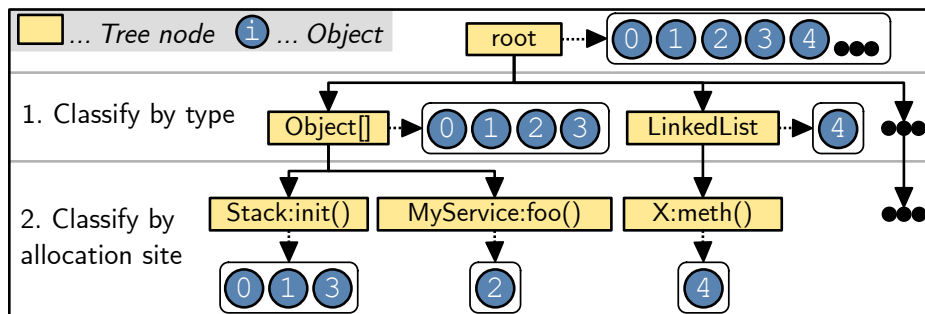


Figure 2: A *memory tree* that first groups all objects by their types and then by their allocation sites.

It is worth mentioning that the order in which the classifiers are applied matters: Applying the *type classifier* before the *allocation site classifier* results in a tree that first groups all objects by their types, and then all objects of the same type by the method they have been allocated in. If we reverse the order of the classifiers, i.e., when we apply the *allocation site classifier* before the *type classifier*, we get a tree that first groups all objects by their allocating method, and then all objects that have been allocated in the same method by their types.

Classifiers: By default, AntTracks currently provides around 30 classifiers out-of-the-box, covering all “standard” heap object properties such as type, allocation site, or allocating thread, but also more sophisticated ones such as age or number of ingoing references. Since object classifiers are just arbitrary functions that return a classification result, new classifiers can be introduced as needed. For example, during our work on data structure analysis, in a matter of minutes we introduced a new classifier that groups heap objects based on their role within a data structure (e.g., head or leaf). This highlights one of the major benefits of memory trees: they are extremely flexible. Depending on the use case or the analysis goal, different combinations and orderings of classifiers can be used and new ones can be introduced if needed. They can be used in user-centered analyses, where the classifier combination is defined by the user, but we also extensively use fixed combinations of classifiers within AntTracks to perform certain automated analyses.

The previously mentioned *type classifier* returns exactly one classification result per heap object, namely its type, thus we call it a *one-to-one classifier*. But there are also other kinds of classifiers. For example, *one-to-hierarchy classifiers* are typically used if a heap object is classified by multiple keys in a hierarchical fashion (i.e., keys with a parent-child relationship). An example is the *call site classifier*, which classifies an object by the chain of methods that called the allocating method, i.e., it returns a call chain that can be merged with other call chains into a tree. The user could then drill down into the resulting tree, exploring the call chain that allocated the most objects. Furthermore, *filters* are a special kind of classifiers that take a heap object as a parameter and return a boolean value whether to include or to ignore the respective object. This reduces the tree size in situations where certain objects are not of interest for the analysis.

Comparison to other heap object groupings: Tools such as VisualVM [221] or MAT [79] support rudimentary queries using the *Object Query Language (OQL)* [4, 49]. Unfortunately, the OQL standard is quite complex, which is why these tools only implement a small part of the whole standard. Furthermore, as an end user, writing OQL is more complicated than combining various classifiers and filters using a comfortable graphical user interface as it is the case in AntTracks.

2.1.2 GC Roots and Closures

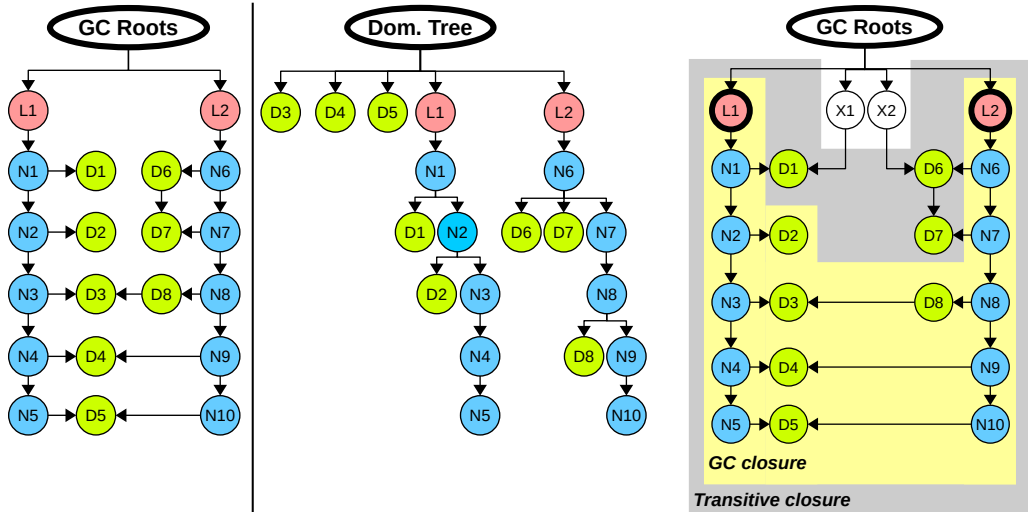
In *Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring* [317], we describe a trace format for recording GC root information, closure algorithms to calculate keep-alive relationships between heap objects, as well as and new metrics and classifiers based on the newly introduced closure concept.

Collecting GC Root Information To analyze keep-alive relationships within the heap and to detect suspicious ownership based on that, two things must be known: (1) the references between the heap objects and (2) information about the various GC roots. While the AntTracks VM was already able to collect the former, we had to extend its trace format to also record GC roots. At the start of each garbage collection, we collect all GC roots from different VM-internal sources and write information about them to the trace file. This information depends on the kinds of the GC roots. For example, for a static field we can extract its class and its field name (e.g., a static field `myList` in class `Buffer`), while for thread-local variables we extract the thread, the class, the method and the variable name (e.g., thread `T1` is currently in method `m()` in class `C`, holding a thread-local variable named `tmp`).

Closures As described above, we need to combine GC root information with information about the references between the objects to analyze suspicious structures. For example, we need algorithms that can highlight those objects that (indirectly) reference a lot of other objects and, more importantly, that keep a lot of other objects alive. We call the set of reachable objects the *transitive closure* [82, 158, 291, 292] and the set of kept-alive objects (or owned objects) the *GC closure*. Thus, the GC closure of a given object `X` expresses its *ownership*, i.e., it contains all objects that could also be reclaimed by the GC if object `X` was garbage-collected.

Existing work in memory analysis focuses on *single-object ownership*, i.e., on detecting single heap objects that own a lot of other objects. While *single-object ownership* makes up a large portion of memory problems, we presents closure algorithms that can also start from a *group* of heap objects to calculate their shared closures. These closures can be used to inspect *multi-object ownership*, i.e., a situation in which objects are kept alive by two or more objects, a problem that is largely ignored in related work.

Traditional single-object ownership analyses typically rely on the *dominator relation* [177] to calculate the GC closure of single objects. This relation has been studied in detail [5, 62, 237] and is used in various domains [68, 83] to analyze domination and ownership. In general, in a rooted directed graph, a node d dominates a node n if *every* path from the root to n must pass through d [252]. Translated to a heap object graph, this means that a heap object d owns a heap object n if every path from a GC root to object n must pass through object d . Using this relation, the object graph of a heap state can be transformed into a *dominator tree*, as shown in Figure 3a. On the left-hand side, an object graph is shown, where each node represents a heap object and the edges represent references between the objects. $L1$ and $L2$ are two singly linked lists with their nodes Ni that point to data objects Di . The right-hand side depicts the object graph’s dominator tree, where the descendants of a node are all owned by that node (single-object ownership). For example, if $L1$ became eligible for garbage collection (by cutting all paths from GC roots to it), seven other objects could be collected, too. Note how $D3$, $D4$, and $D5$ are not placed below other objects in the dominator tree, as they are subject to multi-object ownership, i.e., they are owned by both



(a) Object graph of two singly-linked lists and its dominator tree.

(b) Transitive closure and GC closure for $L1$ and $L2$.

Figure 3: Object graph, dominator tree, and closures of two singly linked lists $L1$ and $L2$.

lists together. Such objects are typically overlooked by approaches that build their analyses upon dominator trees.

Our multi-object closure algorithm differs from dominator-tree-based approaches as it works directly on the heap object graph. Given a certain set of objects to start from, our algorithms can efficiently calculate the transitive closure as well as the GC closure for them. For example, using $L1$ and $L2$ as start objects for the closure calculation, Figure 3b depicts their shared GC closure, i.e., which objects could be collected if both $L1$ and $L2$ were collected, as well as their transitive closure, i.e., which objects are reachable either from $L1$, from $L2$, or from both.

Based on the closures, we can calculate the *deep size* (reachability) and the *retained size* (ownership) for any heap object group S within the object graph. The deep size is the number of nodes (or bytes) in the transitive closure of S , i.e., the number of objects (bytes) reachable from S . The retained size is the number of nodes (or bytes) in the GC closure of S , i.e., the number of objects (bytes) that could be freed by the GC if S died. Knowing which groups of objects keep a lot of memory alive helps the developer to decide which part of the program to inspect in more detail. In the best case, the developer is able to make these heap objects eligible for garbage collection, allowing the garbage collector to also collect all objects in their GC closure. Since every tree node in a memory tree represents a group of heap objects, our memory trees and closure algorithms complement each other well, as we now can calculate the deep size and the retained size for each tree node.

2.2 Data Structure Analysis

In *Analyzing the Evolution of Data Structures in Trace-Based Memory Monitoring* [316] as well as in *Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection* [318] we present an approach to describe, detect and track growing data structures. If the monitored application contains a possible memory leak that is caused by a growing data structure, our approach is able to automatically report that data structure. The user can then use AntTracks’s flexible classification mechanism, together with new classifiers for data structure analysis, to inspect the suspicious data structure in detail.

In general, a data structure is described as *a collection of data values, the relationships among them, and the functions or operations that can be*

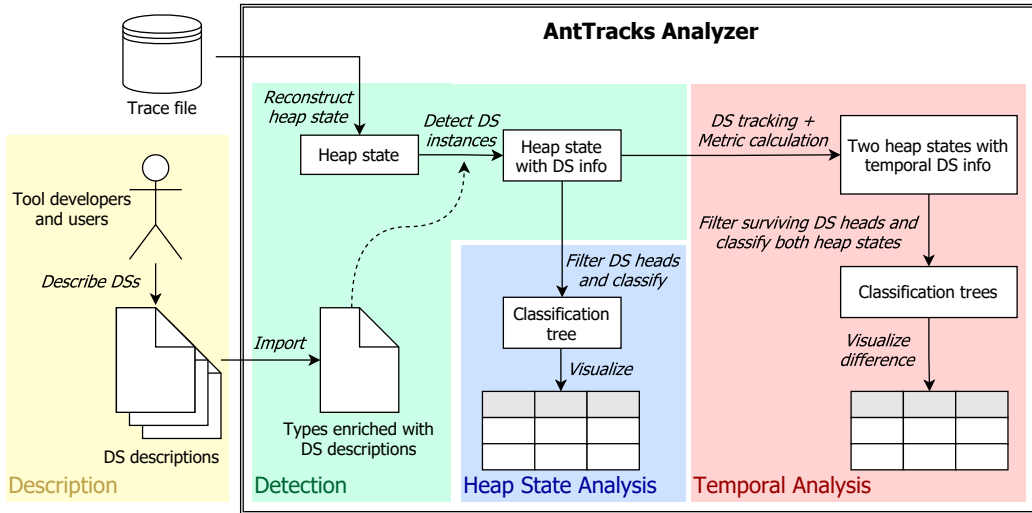


Figure 4: Our data structure analysis approach consists of four stages: (1) *Description* of data structures using a DSL, (2) *detection* of data structures in reconstructed heap states, (3) *heap state analysis*, i.e., data structure analysis at a single point in time, and (4) *growth analysis*, i.e., tracking data structures over time, detecting those with suspicious growth.

applied to the data [312]. While certain memory leak analyses focus on the operations performed on data structures [350, 352] (e.g., by comparing the number of insert operations to the number of remove operations), we are interested in the relationships, i.e., the references between the objects, that make up a data structure. Our approach uses a domain-specific language (DSL) developed in CocoR [203, 344] to define the shape of data structures. This DSL allows us to define data structures in terms of a *head* element (e.g., an object of type `HashMap`), *internal* elements (e.g., objects of type `HashMap$Node`), and *leaf* elements (e.g., the keys and values stored in a hash map node). As shown in Figure 4, these descriptions enable us to detect data structures in reconstructed heap states. Once detected, we can inspect them at a single point in time (heap state analysis) or we can track them across their lifetime to monitor their growth (temporal analysis). For temporal analysis we can calculate a data structure’s transitive closure, GC closure and a newly defined data structure closure (which includes all objects that directly belong to the data structure) at multiple points in time. Based on the growth of these closures, we can detect which data structures grew

the most, while at the same time provide information about the kind of growth. For example, certain combinations of retained size growth (i.e., ownership growth) and data structure size growth (i.e., new objects added to the data structure) allow us to draw conclusions about the underlying problem pattern. If we have a continuously growing data structure, but the data structure’s ownership does not grow, this clearly indicates that at least a second data structure is involved in keeping the accumulating objects alive. Using this information, the user may then perform further analysis steps to inspect this multi-object ownership in more detail. Similar conclusion can be drawn for other combinations of growth metrics, and for each combination different subsequent analysis steps may be suggested.

2.3 Visualization

The concepts discussed so far concerned either data structures or algorithms to perform a variety of memory analyses. Even though these are important topics, it is also important to display the gathered information in a way that allows the user to easily digest it. This is even more true since most memory analysis tools currently lack advanced visualizations, even though data visualization [115] can help to convey information faster [149, 309]. It can facilitate the identification of patterns and relationships [204, 310] which can lead to new insights [310].

In this thesis, memory trees and their growth are a recurring theme. Interpreting system growth through visualization is a widespread research topic. For example, in the domain of software evolution and program comprehension various studies have shown that using graphical means can help users in understanding and interpreting software system growth [31, 64, 86, 88, 89, 250, 251, 339]. Thus, we present three visualization techniques that we implemented to support users in analyzing memory evolution and memory growth over time.

2.3.1 Drill-down Trend Visualization

In *AntTracks TrendViz: Configurable Heap Memory Visualization Over Time* [325], we present an approach for visualizing the evolution of memory trees over time. To this end, we use a time-series chart that implements a drill-down feature. Basically, the chart shows the evolution of the heap

memory grouped by a single heap object property, for example how many objects of different types existed over time. The drill-down feature allows the user to select a suspicious object group by clicking on its series in the chart, which then opens another chart that displays more detailed information about the selected objects, e.g., where the objects have been allocated.

We chose time-series plots as a means of visualization since they are well-known, easy to understand and one of the most frequently used types of visualization in statistics [300]. In general, each series in a time-series plot takes the following form: $D = \{(t_1, y_1), (t_2, y_2), \dots, (t_n, y_n)\}$ [311], i.e., it consists of data pairs where a given point in time t_i has a certain numeric value y_i assigned to it.

We based our TrendViz visualization on a list of memory trees that are created according to a user-defined combination of classifiers. For example, at every garbage collection point, we may group all live objects first by their types, and then by their allocation site. This results in a data set $D = \{(t_1, tree_1), (t_2, tree_2), \dots, (t_n, tree_n)\}$, i.e., for each GC point we have a corresponding memory tree, where each tree has two levels: On the first level, the tree is split based on types, and on the second level each type is split by allocating methods.

In our visualization, initially only the evolution of the first level of the memory tree is shown. For example, if we classified all objects by types and allocation sites, a single time-series chart would depict the evolution of the objects grouped by type, showing each type as a separate series. If a certain series (e.g., type) gains the user’s attention, for example because it exhibits strong growth, the user can select it for drill-down. This opens a second time-series plot below the first one, which depicts the memory evolution of the selected node’s children, i.e., the evolution on the second tree level. The example in Figure 5 shows objects of type `Date` that accumulated over time (see top chart). The user then selected this object group (highlighted in yellow), and a second chart opened below the current one, displaying the selected type’s allocation sites over time. This way users can interactively collect information about suspicious objects accumulating over time.

The individual time-series plots can be adjusted in various ways. For example, a maximum number of shown series may be defined, the shown metric (objects or bytes) can be switched as well as the kind of the shown size (shallow size, deep size, retained size, or data structure size). The visualization can also be switched between a stacked area chart (mostly useful for the shallow size) and a traditional line chart (more useful for sizes such as

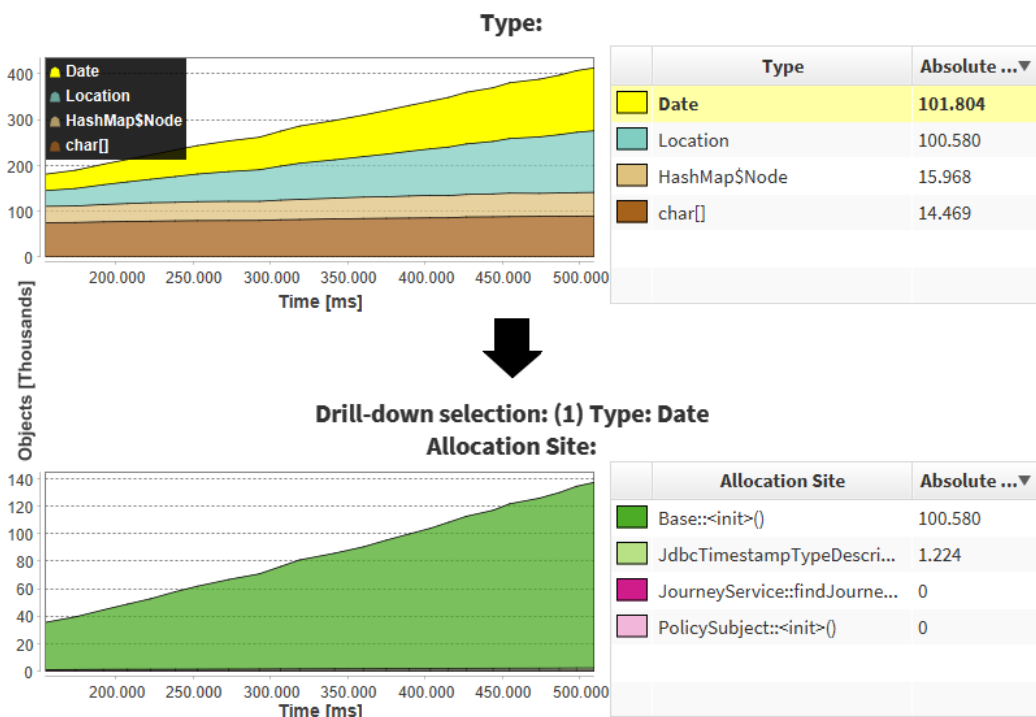


Figure 5: The AntTracks *TrendViz* displays the heap evolution grouped by a selected classifier combination, allowing users to drill down into object groups by clicking on them.

the retained size). In stacked charts, also the series' sort order may be configured (e.g., ordering the series from top to bottom based on their absolute growth, as done in Figure 5). This enables users to generate sophisticated drill-down time-series. For example, they may filter for data structures and show the data structures with the strongest growing retained size as separate line chart series. This highlights those data structures that exhibit the strongest single-object ownership. The user may also use a second classifier for drill-down, e.g., the *data structure leaf classifier*, which groups all leaves inside a data structure based on their types. A drill-down into a suspicious data structure would thus open a second time-series plot that would highlight which object types accumulated the most over time in the selected data structure.

In AntTracks, we implemented this feature using JFreeChartFX [103, 163], a modified version of JFreeChart [102] for JavaFX.

2.3.2 Memory Cities

In *Memory Leak Visualization using Evolving Software Cities* [327] and *Memory Cities: Visualizing Heap Memory Evolution Using The Software City Metaphor* [326] (Best Paper Award), we present our *Memory Cities* approach that visualizes an application’s *heap memory evolution over time* using the *software city* metaphor.

General approach: Traditionally, the software city metaphor visualizes software artifacts (e.g., classes) as buildings that are arranged in districts (e.g., packages), where the size of a building conveys some kind of information (e.g., the classes’ LOC). While the software city metaphor is typically used to visualize *static* artifacts of a software system such as class hierarchies, we use it to visualize the *dynamic* memory behavior of an application. In our approach, we group the heap at multiple points in time, e.g., at each garbage collection point, using a user-selected combination of classifiers, similar to our *TrendViz* approach. For example, at each garbage collection, we may group all heap objects first by their allocating threads and then by their types. For each of these memory trees, we may then create a corresponding city layout. In the given example, threads (the first level of the tree) would be visualized as districts. Within these districts, buildings for the different types (the second level of the tree) are placed. Continuously updating the city over time, i.e., moving from one point in time to another, creates the engaging feeling of an evolving city as buildings and districts grow and shrink over time. Our visualization can be used to inspect an application for memory leaks by searching for suspicious growth behavior, i.e., suspiciously strongly growing buildings. For example, if the building `Buffer` (type) in district `T1` (thread) grows over time we can conclude that the thread `T1` allocated more and more objects of type `Buffer` that accumulated over time, which is a typical memory leak behavior. We further support memory leak analysis by also using other visual attributes beside building size. For example, strongly growing buildings are highlighted in color, and less suspicious buildings can be made semi-transparent to guide the user’s focus to more important buildings, as can be seen in Figure 6.

To transform a memory tree into a city layout, we use a tree map algorithm [138, 257, 268], more specifically the *squarified tree map* algorithm by Bruls et al. [41]. A tree map layout places rectangles (in our case approximate squares) within each other to model the tree hierarchy, and the area

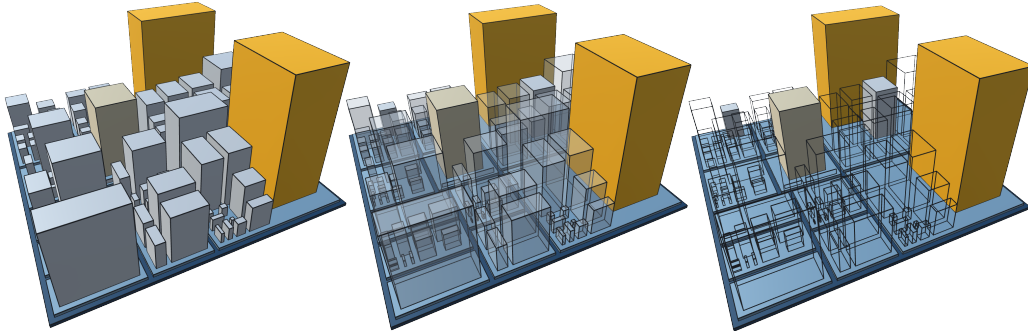


Figure 6: Three different city visualizations. Left: Every building is shown fully opaque. Middle: The five strongest growing buildings are shown fully opaque, the rest is set to 40% opaqueness. Right: The five strongest growing buildings are shown fully opaque, the rest fully transparent (except for outlines).

sizes of the rectangles indicate one of the tree nodes' values. For example, if the rectangles B and C are placed within rectangle A , B and C are children of A , and larger rectangles represent larger values (in our case more heap objects). Even though a tree map can be displayed as a 2D visualization on its own, a third dimension *height* can be added to each rectangle to create a 3D visualization [30], similar to a city. By default, in our memory cities, both the base area and the height represent the same metric, i.e., either object count or byte count. Mixing these metrics, i.e., using one metric for the base area and the other one for the height, is possible but may not yield visually appealing results. For example, having a node that represents few very large arrays could result in (a) extremely narrow buildings that are quite tall or (b) extremely wide buildings that are quite flat. Such unrealistic building sizes would distort a realistic city feeling and may also be hard to interact with in certain situations (e.g., narrow tall buildings are hard to see and click). A possible solution to this could be to use the object count as-is for one visual attribute and the byte count as *categorical data* for the other, e.g., mapping the byte count to a few fixed heights such as tiny, small, medium, large and huge.

Another challenge when working with evolving tree maps is layout stability [112, 258, 273]. Layout stability means that the layout, i.e., the placement of the rectangles / buildings should not change between two points in time. When looking at visualizations, humans build *cognitive maps* that are based on spatial relationships and attributes of the presented data [155]. Thus,

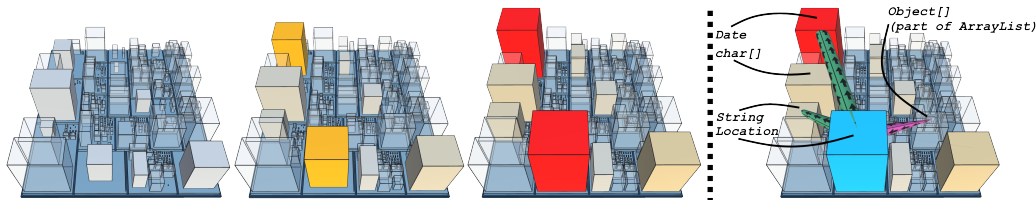


Figure 7: In this application, `Location` objects accumulate over time (selected blue building), together with many `Date` objects and a few `String` objects that are referenced by the `Location` objects.

if these relations (e.g., positions) change, the user may lose track of certain buildings and consequentially cannot follow their evolution over time. In our work, we use the *static position animation* [171] approach to achieve a stable layout over multiple points in time. In this approach, we calculate a general layout for the city *once* using the tree map algorithm to reserve space for all buildings based on the maximum area sizes they may reach. Then, when visualizing the city for a given point in time, the districts and buildings are just centered within their reserved area based on their current size.

Another feature of our memory cities is its *reference inspection*. Once a suspiciously growing building is detected, typically the next question is “*What causes this growth?*”. As explained, objects accumulate over time because they are directly or indirectly referenced by GC roots. Thus, memory cities allow the user to select a certain building to display its outgoing and incoming references as colored cones between buildings. Incoming cones (pink) describe the *is-kept-alive-by* property; outgoing cones (green) describe the *keeps-alive* property. This is important information to find out which other objects cause the accumulation of a certain object group. Figure 7 shows an example of an evolving software city where growing buildings change from gray to red (left). It also shows how the reference inspection of a given building may look like for a selected building (right, selected building highlighted in blue). Based on the gained information about growing object groups and their references to and from other objects, the user should have a rather clear picture of where to inspect the source code to fix a potential problem.

Metaphors and Software Cities in Other Domains The idea of using metaphors [168] in visualization is not new. A few examples are geometrical figures as a metaphor for coding constructs to teach programming [143],

3D structures to visualize database query results [39] (using attributes such as size and position to convey information), colored virtual reality tunnels to analyze concurrency in programs [244, 245], or data visualization in the form of maps [121]. Knight and Munro [156, 157] strongly promoted the use of metaphors for software visualizations, especially their metaphor of a *software world*. As an alternative to software worlds, 3D city visualizations [224] emerged which, as studies show [46, 183], are used in a variety of different domains. Application domains include the visualization of software quality metrics [32, 169–171], evolution visualization of large-scale software system [333, 335–339], software development history visualization [284–286], concurrency visualization [306], software component communication and dependency visualization [89–93], software performance visualization [196, 210], business process visualization [254], test case analysis [279–281], and they have been used in virtual reality [87, 196, 250, 251] as well as within computer games such as Minecraft [12–15]. To the best of our knowledge, we are the first to use the software city metaphor in the domain of memory monitoring, and also extended the basic idea of software cities with features such as *reference inspection*.

We developed our 3D memory city visualization using *Unity* [220]. Even though Unity is mainly used as a game engine, it more and more also qualifies as a general-purpose real-time 3D development platform [343].

2.3.3 Tree Visualizations

In *Heap Evolution Analysis Using Tree Visualizations* [328] and *Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations* [330] (Best Paper Award), we present a new visualization approach that uses well-known 2D tree visualizations in combination with novel interaction features to convey information about an application’s *heap memory evolution over time*.

Similar to our *Memory Cities*, which use a 3D city representation to visualize the evolution of memory trees, we also explored how well-known 2D tree visualizations can be leveraged to ease memory analysis. In the past, it has been shown that tree visualizations are useful for a variety of analysis tasks [24, 294, 308], yet their application in memory monitoring is rare. Thus, we first performed an analysis of existing tree visualization techniques [261] based on a requirements catalogue to select approaches that are suitable for interactively visualizing heap memory evolution. For example, we excluded

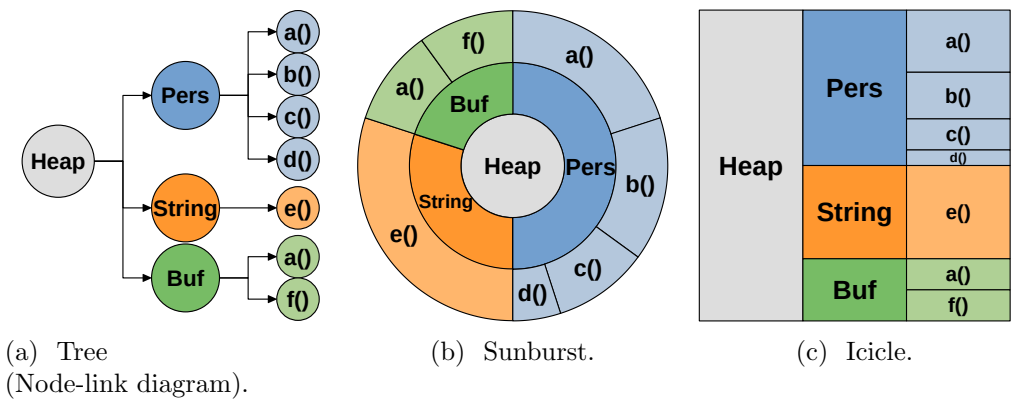


Figure 8: Three visualizations showing the same data.

approaches that involve complex layout phases [9, 359] that may cause the application to exhibit inconvenient lag, an important property of interactive tools [44, 185]. We also surveyed existing studies on the usefulness and visual appeal of tree visualizations [17, 50] and finally selected sunburst plots [6, 59, 283] as well as icicle plots [11, 115, 164] as our means of tree visualization.

In Figure 8, three different tree visualization techniques are shown that depict the same data. In a node-link diagram (Figure 8a), the tree is visualized explicitly, i.e., the tree nodes are explicitly linked via edges [262]. Typically, these kinds of visualizations do not use the node size to convey information, rather the nodes are equal in size or scaled to fit their text. On the other hand, the *sunburst plot* (Figure 8b) and the *icicle plot* (Figure 8c) use variable-sized graphical elements to visualize tree nodes, i.e., larger graphical elements represent larger values in the respective tree nodes (for example, more heap objects). The sunburst plot uses ring segments to visualize tree nodes and the tree hierarchy is moving outwards, starting at a *root circle* in the middle. The icicle plot uses rectangles as a means of visualization. In a horizontal icicle plot all rectangles have the same width, and the underlying tree node values are mapped to the rectangles' heights. The tree hierarchy is moving from left to right, starting at a root rectangle on the left that spans the whole height. For example, both the sunburst plot and the icicle plot in Figure 8 show that 50% of the heap is occupied by objects of type **Pers**, and most of these objects have been allocated in method **a()**.

The overall visualization approach and the order of operations is similar to our memory cities approach, yet it has some relevant differences. Similar

to memory cities, the user first selects a combination of classifiers based on which the heap is grouped at multiple points in time, which results in a list of memory trees. Then, each of these trees is mapped to its respective visualizations, i.e, to a sunburst plot as well as to an icicle plot. To ensure layout stability, before mapping a tree to its visualization, we apply a user-selected sorting strategy, e.g., sorting based on the absolute growth, on all tree nodes to ensure as few spatial changes in the visualization when going from one point in time to another.

What distinguishes our approach from others is that we provide two ways to visualize the trees' *evolution over time*. A screenshot of our tool can be seen in Figure 9. In our *timeline-based* visualization (5), a time-series chart depicts the overall memory consumption over time. For each data point, the user can toggle the display of a corresponding memory tree visualization above the chart. These memory trees are shown side-by-side, which allows the user to inspect the evolution of the heap over time by comparing the different heap states. In our *time-travel-based* visualization, two space-filling tree visualizations show the monitored application's heap memory at a given point in time. (1) visualizes the complete memory tree with all its branches and levels. By clicking on a node, users can select a *drill-down node* (shown as a hatched segment in (1)) which is used as the root node in (2) to depict the selected tree branch in more detail. Clicking on the root node drills out again. Further, both visualizations are synchronized, e.g., the hovered node in (2) (red border) is also highlighted in (1). Users can step back and forth in time using buttons or a slider (3), causing the visualization to update itself. Also, the visualization can be switched between sunburst plots and icicle plots, the depicted metric can be switched between number of objects and number of bytes, and the nodes' sort order can be changed (4). Both of our visualization techniques should help users to gain new insights about the memory behavior of their application and to detect (problematic) memory trends.

This web application has been developed using d3.js [37, 38, 45], with a focus on usability and visual appeal. For example, when switching between two points in time, the visualization uses tweening [341], i.e., smooth transitions, to make it even more easy for users to track the development of tree nodes over time.

2.4 User Guidance and User Behavior

During tool demonstrations and small-scale studies, we recognized that detecting and inspecting memory anomalies can be a non-trivial task, even with advanced tools. This is especially true for novice users who have no

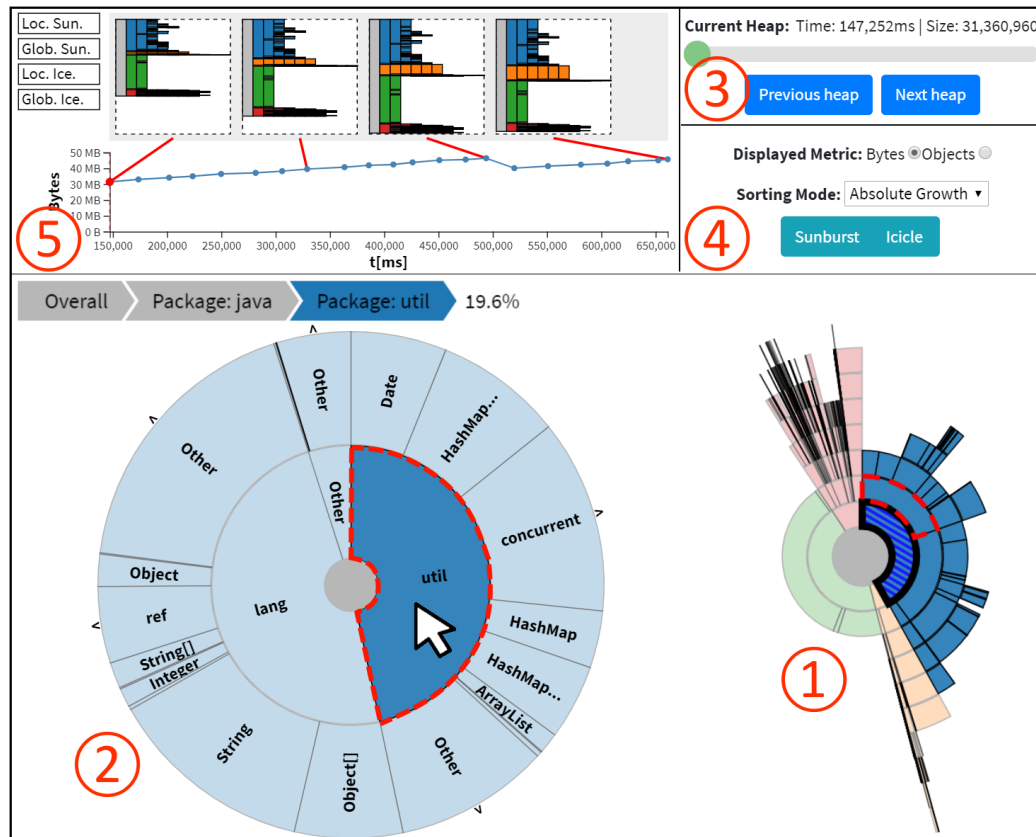


Figure 9: Overview of our visualization tool. (1) and (2) show the *time-travel-based* visualization, i.e., both show the heap at the same point in time. (1) displays the complete memory tree with all branches and levels. Clicking on a tree segment selects a new *drill-down node* (hatched segment in (1)) which is used in (2) as root node to inspect a certain branch in more detail. Both visualizations are synchronized, e.g., the hovered node in (2) is also highlighted in (1). On the top right, the (3) time controls and the (4) visualization options can be found. Beside them, the (5) *timeline-based* visualization is placed, which can show multiple trees side-by-side.

experience in memory monitoring. Thus, this thesis also covers topics such as how users behave during memory analysis (which we investigated through a detailed user study) and how memory analysis tools can be designed to make the analysis easier for novice users.

2.4.1 Automatic Detection of Suspicious Time Windows

In *Detection of Suspicious Time Windows in Memory Monitoring* [319], we present our first step towards automatic user guidance by automatically detecting time windows during which a monitored application behaved abnormally with regard to memory consumption. For example, continuous memory growth may be the result of a hidden memory leak, while a strong fluctuation in memory consumption may be the result of high memory churn. Without guidance, it is up to the users (1) to know how suspicious memory behavior looks like, and (2) to detect suspicious time windows on their own.

To support novice users, we developed three algorithms that inspect the memory evolution of an application as time-series data [96]. They are able to recognize time windows with suspicious (1) continuous memory growth, (2) high GC overhead, or (3) high memory churn, i.e., strong fluctuations in memory consumption. To achieve this, they use regression [206] and heuristics that mimic human behavior. For example, when searching for a possible memory leak, we employ a heuristic that looks for the longest time window

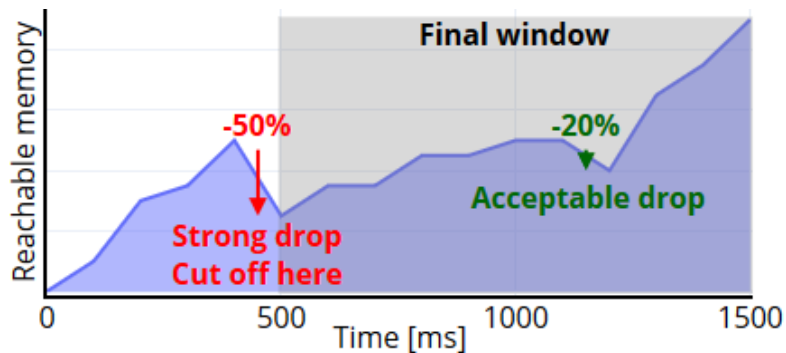


Figure 10: The heuristic-based memory leak detection algorithm, one of the three algorithms we devised, searches for the longest window that does not contain strong drops in the reachable memory.

that does not contain a strong memory consumption drop until the end of the trace (see Figure 10). This time window is then checked against certain thresholds such as a minimum window length and a minimum memory increase. If a suspicious time window has been detected, it is highlighted to the user and can then be inspected in more detail using the other analysis techniques presented in this thesis.

2.4.2 Cognitive Walkthrough and User Study

In *Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study* [321], we assessed the usability [1, 161] of our tool *AntTracks Analyzer*. Except for a single study [357], we were not able to find existing work on the usability of memory monitoring tools, as nearly all existing studies evaluate approaches with a focus on their performance overhead.

Cognitive Walkthrough First, we performed a systematic cognitive walkthrough based on the *Cognitive Dimensions of Notations framework* [26, 27, 29, 106–108]. This framework offers a vocabulary for discussing usability issues and their trade-offs, including a set of *cognitive dimensions* (CDs), which are summarized in Table 3. It has already been

Table 3: Cognitive dimensions used for the walkthrough (taken from [107]).

Dimension	Description
Abstraction	types and availability of abstraction mechanisms
Closeness of Mapping	closeness of representation to domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Error-proneness	notation invites mistakes
Hard Mental Operations	high demand on cognitive resources
Hidden Dependencies	important links between entities are not visible
Premature Commitment	constraints on the order of doing things
Progressive Evaluation	work-to-date can be checked at any time
Provisionality	degree of commitment to actions or marks
Role-expressiveness	the purpose of a component is readily inferred
Secondary Notation	extra information in means other than formal syntax
Viscosity	resistance to change
Visibility	ability to view components easily

successfully used in the past to assess software tools [20, 162, 189, 232, 235], visual diagrams [28], temporal specification notations [166, 167], or visual modeling languages [67, 340]. Thus, we decided to use it as a basis for our walkthrough, in which we evaluated each view of our AntTracks Analyzer tool by checking it against the various cognitive dimensions.

Three assessors, two developers of AntTracks and an experienced colleague, used AntTracks and systematically rated its views with regard to fourteen different cognitive dimensions. Each assessor rated each cognitive dimension on each view using a color-based three-level classification: (1) green: no issues found, (2) yellow: room for improvement, and (3) red: serious flaws. In addition to that, every assessor also took textual notes regarding each cognitive dimension on each view. The results of these assessments were then merged during a discussion session to ensure a common understanding of all raised concerns. Figure 11 shows the main result of our cognitive walkthrough: a table with merged evaluation results and detailed textual assessor notes. Based on this table, serious design flaws were immediately fixed in AntTracks, and it was also used as a decision tool on what to focus on during a subsequent user study.

User Study Once serious usability flaws that were revealed during the cognitive walkthrough had been fixed, we designed a user study to investigate findings from the CD assessment in more detail. We relied on well-established guidelines for the design of the study tasks [159], the subject selection [127], as well as for conducting empirical studies [253].

As system under inspection, we chose the web shop application JPet-Store 6 [205], since it is complex enough to mimic a real application, yet small enough to be easily explained using a single UML diagram [36]. It has also been used in many other studies [91, 136, 147, 148, 307]. We seeded the system with two defects: one memory leak by keeping shop items alive, and one memory churn hotspot by using Java streams inefficiently.

Software engineering students from our university used AntTracks to investigate the memory evolution of this application, and we observed them while they tried to detect and fix these anomalies by solving five given tasks. While some of the participants reported that they had used various tools to inspect memory before [79, 104, 111, 133, 187, 207, 212, 216, 221, 296], all of them classified themselves as novices. Also, we made sure that the participants had not used AntTracks before.

Task	Detection: Memory Growth	Inspection: Evolution over Time	Inspection: Single Point in Time	Inspection: Single Point in Time	Detection: Memory Churn	Inspection: Evolution over Time	Cross-Cutting	
AntTracks View	Overview	TrendViz View	Heap State View	Graph View	Details View	Short-living Objects View		
Cognitive Dimensions	Abstraction	Overview uses easy terminology.	Abstraction into chart series -> improve by ...	Maybe terminology? Data structure DSL ...	Nodes represent groups of objects -> understandable? ...	GC chart	Is the content of the tree view clear?	Terminology, icons, etc.
	Closeness of Mapping	GC chart	Drill-down feature may not be clear. The hierarchical ...	Tree visualized as hierarchical TreeTableView ...	How to display different elements (Objects, GC roots, ...)	GC chart	Tree visualized as hierarchical TreeTableView ...	
	Consistency	Evolution data is by default presented as charts in AntTracks.		Hierarchical data is by default presented as TreeTableView ...	To achieve immersion and closeness of ...		Other column names than on heap state view.	Are there annoying inconsistencies?
	Diffuseness		Overcrowded classifier selection, also see viscosity.	Classifier selection is too complex. Highlight most ...	Test that not too many different notations are used, ...	Explanatory text is too long.	Many charts on overview - too many?	Unnecessary or unnecessarily complex views?
	Error-proneness	Possible flaw: Chart interaction. Positive: Zoom ...		Operations in context menu clear? User-defined ...	Make sure that operations that would create too ...	See Overview (Chart interactions)		
	Hard Mental Operations	Do users recognize growing memory as problem?	See abstraction & closeness of mapping.	User is free to use any classifier combination. Certain ...	Even though users can inspect graphs, the detection of ...	Interpretation of charts hard?	Normal classification trees.	
	Hidden Dependencies	Zoom is synced, selection is synced.	Highlight selection in parent chart better. Also display ...	BUG: New classification in heap state may ...			Link from pie chart to table clear?	Are there any dependencies that we did not find yet?
	Premature Commitment		Time window has to be selected beforehand ...	Time has to be selected beforehand	Time has to be selected beforehand. Once nodes are ...			Order of operations, etc.
	Progressive Evaluation	User can check how many of the suggested time ...	Selected value is shown for every level. The more levels, ...	Position withing classification tree determines progress.	User can always check the path he/she has already ...			
	Provisionality	Can open a new heap state without problems, can ...	All settings can be changed arbitrarily.	Abortion of long running operations is possible.	View is always resettable. Future work: "What-if"-games.			
	Role-expressivness	Memory chart clear. GC chart probably not directly clear.	Is it clear what a single chart is showing?	Should be clear, ask if the tree table visualization was ...	Are the different types of nodes clear?	Charts maybe not clear, check if users understand what ...	Do users understand the charts?	
	Secondary Notation							
	Viscosity		Inflexibility of the classifier selection. Classifiers cannot ...	Order of classifiers cannot be changed using drag-and-...	Graph grows rather fast.		Order of classifiers cannot be changed using drag-and-...	
	Visibility	New overview tab was implemented: Now Memory + GC ...	Drill-down feature has been improved (with table, etc.) ...	Should be clear, ask if the tree table visualization was ...	Legend was needed.	Many charts at once, may be overwhelming.		Tab system. Do users find out ...

Figure 11: Such a table was used to document and classify the results of the CD assessment. Each column represents an memory analysis tasks performed on one of AntTracks' views. Each of the 14 cognitive dimensions [26] is shown in a separate row. Green cells indicate cognitive dimension for which no issues were found on the respective view, yellow cells highlight cognitive dimensions that may be improved on the respective view (i.e., these dimensions were possible subjects for a more detailed evaluation in the user study), and red cells indicate serious problems that had to be fixed before the user study. The texts in the cells show parts of the notes that were taken by the assessors during the walkthrough. Certain view-CD-pairs (cells with black text and thick border) were chosen to be inspected in more detail during the user study.

We asked the participants to ‘think aloud’ [125, 132, 209] during their session, i.e., to describe what they were doing and to comment on any concerns. After they finished the given tasks, each participant was also interviewed on the usefulness of the tool [69] and completed a usability questionnaire [208].

Once all study sessions were finished, we labelled all observations, statements and interview answers to allow their systematic use. For example, the think-aloud statement on AntTracks’ overview screen ‘*In the chart, I can see that my memory grows more and more, that is not good.*’ received the labels ‘*Detects Growth In Chart*’ as well as ‘*Recognizes Growth as Problem*’. For this, we adopted and adjusted an iterative labelling process [101] that is similar to Open Coding [264]. Based on a common set of labels (that was initially developed in a joint meeting), three coders individually coded all observations and statements. In the case that an observation or statement could not be mapped to an existing label, the coders collectively decided if a new label should be introduced. In a final joint discussion, the coders merged their three individually labelled lists of observations and statements into a single list, thereby discussing and resolving possible differences.

A common request across multiple participants was ‘better guidance’. When asked, the participants confirmed our observations that guidance would have especially been needed in the following tasks: (1) deriving knowledge from the displayed information, i.e., interpreting the data; (2) selecting suitable next analysis steps based on the findings; and (3) guidance within the IDE to help fixing the problems, instead of just finding their locations in the source code.

Based on the overall results of the user study, we derived nine recommendations for memory tool developers. Based on these recommendations, we implemented various improvements in the AntTracks Analyzer to increase its usability for novice users. This ranged from minor changes such as improved chart interactions (to better match existing tools with similar functionality [10, 178]) to fully-fledged user guidance, as will be discussed in the next section.

2.4.3 Guided Exploration

Our paper *Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools* is currently under review at the 13th Conference on Engineering Interactive Computing Systems (EICS).

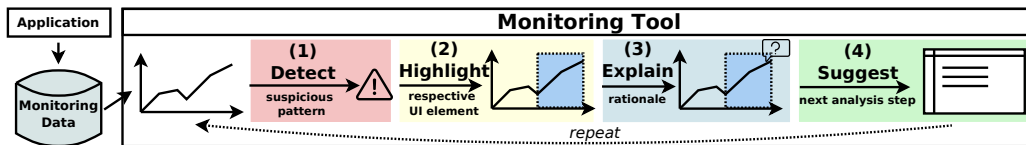


Figure 12: The four steps of guided exploration: (1) Detection, (2) Highlighting, (3) Explanation, and (4) Suggestion.

The previously outlined cognitive walkthrough and the user study made it clear that novice users are easily overwhelmed when they are confronted with complex memory analysis tools that focus on expert users. According to our experience, being too complex for novices is a widespread problem across (memory) monitoring tools. Thus, we developed a guidance method called *Guided Exploration* that should help tool developers to integrate user guidance into their monitoring tools. Such guidance increases the learnability and ease-of-use of a tool and particularly supports novice users, i.e., users with little or no background in the tool’s domain. This is an important goal, since factor such as learnability have a strong influence on tool usage and user attraction [249], and the use of monitoring and analysis tools can lead to higher software quality [3, 236, 282, 299].

The idea of user guidance is not new. Folmer and Bosch [95] classify two general *guidance patterns* that are typically used to increase tool usability: (1) *wizards* [74, 297, 358], i.e., a linear series of dialog views that ask a number of questions to then automate certain tasks, and (2) *context-sensitive help* [74, 194, 287], e.g., systems providing “how to” information or suggestions during program use as part of an intelligent user interface [116, 131, 140, 193]. Context sensitive help is often applied in micro-learning and gamification approaches [105, 113, 128] to improve the *onboarding* experience [246], i.e., while introducing a person to a new tool to improve the person’s success using it [274]. To the best of our knowledge, we are the first to describes a general guidance method in the context of interactive monitoring tools, especially in the domain of memory monitoring.

In general, when following our guided exploration method, tool developers should first identify their tool’s most important analysis tasks by building a task model [99, 160, 184, 186, 195, 225]. For each step in these analysis tasks, the tool then should provide four support operations, as shown in Figure 12: (1) First, the tool should automatically *detect* potential problems, i.e., suspicious patterns (for example by using automated time series analy-

sis [19, 96, 179, 206, 260, 313] or by using algorithms inspired by recommender systems [84, 240]). (2) To help users understand on what the suspicion is based, the respective user interface elements should be *highlighted* [180, 345]. (3) Since users may require background knowledge to comprehend the highlighted information, *explanations* should be provided why the highlighted information is interesting [58, 139]. (4) Finally, based on the detected problem, subsequent analysis steps should be *suggested*. By providing these support operations, tools *guide* users through the whole analysis process, helping them to *explore* the data until the root cause of a problem is found. At the same time, being guided by the tool results in a learning-by-doing effect [256], i.e., users learn about the capabilities of the tool and how to use them efficiently.

In addition to presenting guided exploration as a general concept, we focus on its application in the domain of memory monitoring. We present two complete guidance processes for (1) memory leak analysis as well as for (2) memory churn analysis, two of the most common analyses in memory monitoring tools. To demonstrate how guided exploration can be implemented, we describe how we refactored our memory monitoring tool AntTracks Analyzer to support this concept. In two user scenarios on different applications, we resolve memory problems by just following the suggestions of the tool and show how users profit from AntTracks’s new guidance feature.

We also discussed the applicability of guided exploration with authors of a lock contention monitoring tool [122, 259, 263]. As we received positive estimations of its applicability there, we are confident that guided exploration can also be useful for other interactive monitoring tools outside of the domain of memory monitoring [233, 234]. Thus, our work also encompasses a detailed discussion of current limitations and future work and outlines how guided exploration could be further improved and refined in the future. We hope that our contribution will motivate and help other monitoring tool developers to improve the learnability and accessibility of their tools.

2.5 Memory Churn

As a last topic in this thesis, we tackled another common memory anomaly besides memory leaks: high memory churn. High memory churn is also known as *excessive dynamic allocations* [227] or *high allocation density* [75] and is a common performance anti-pattern [270–272]. As its name suggests,

high memory churn is the result of frequent allocation and collection of heap objects. It negatively affects an application’s performance, since it takes time to allocate the objects, as well as to collect them during garbage collection. Even though modern garbage collectors mostly reclaim objects concurrently to the application’s execution [61, 71, 94, 141, 181, 214], nearly all of them require a *stop-the-world* pause at some point [182, 301], i.e., the application is halted while the GC is running. Such GC pauses can make up a significant portion of the application’s run time.

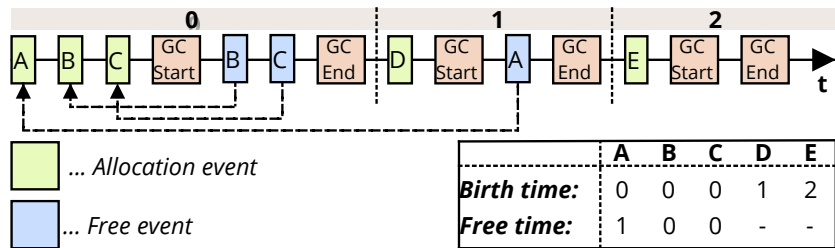


Figure 13: For every heap object, its *Birth Time* and *Free Time* is reconstructed, which is then used to calculate the object’s *lifetime*. Objects that are allocated and freed during a memory churn hotspot are the hotspot’s major contributors and should be investigated.

In this work, we present a novel approach to support developers in investigating high memory churn. We rely on our work on detecting suspicious time windows [319] to detect *memory churn hotspots*, i.e., time windows where unusually high amounts of garbage are produced and collected. Since most objects are short-living [66, 182], we want to draw the user’s attention to objects that died young, i.e., to those that died without surviving a single garbage collection. Thus, if our algorithm detects a memory churn hotspot, we calculate the *lifetime* of each object that was garbage-collected within that hotspot. This lifetime information can be reconstructed from the memory trace, as we know between which GC runs each object was created and during which it was collected, as shown in Figure 13. Since most objects are allocated in relatively few methods [142, 305], we group all garbage-collected objects by their lifetime and allocation site, which highlights those methods that cause the most short-living garbage. Using our flexible heap object classification mechanism and visualizations, the user can also use other grouping combinations to inspect the memory churn hotspot’s garbage. This enables the users to pinpoint object types and code locations that should be in-

spected to reduce memory churn. To reduce the number of allocations, we suggest to reuse existing objects [129, 190], for example by implementing a caching strategy and/or by using design patterns such as the *prototype pattern* or the *flyweight pattern* [97].

Part II

Publications

Chapter 3

Memory Traces and Their Processing

3.1 Heap Object Classification and Multi-Level Grouping

This section includes the paper [331] that describes our classification and multi-level grouping approach, the core of the AntTracks Analyzer. Most of our analyses and visualization approaches rely on aggregated memory trees produced by this method.

Paper:

Markus Weninger, Hanspeter Mössenböck:

User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE 2018, Berlin, Germany, April 09-13, 2018. - **Best Paper Nominee**

User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring

Markus Weninger

Institute for System Software
Christian Doppler Laboratory MEVSS
Johannes Kepler University Linz, Austria
markus.weninger@jku.at

Hanspeter Mössenböck

Institute for System Software
Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Software becomes more and more complex. Performance degradations and anomalies can often only be understood by using monitoring approaches, e.g., for tracing the allocations and lifetimes of objects on the heap. However, this leads to huge amounts of data that have to be classified, grouped and visualized in order to be useful for developers. In this paper, we present a flexible offline memory analysis approach that allows classifying heap objects based on arbitrary criteria. A small set of predefined classification criteria such as the type and the allocation site of an object can further be extended by additional user-defined criteria. In contrast to state-of-the-art tools, which group objects based on a single criterion, our approach allows the combination of multiple criteria using multi-level grouping. The resulting classification trees allow a flexible in-depth analysis of the data and a natural hierarchical visualization of the results.

KEYWORDS

Memory, Monitoring, Analysis, Tool, Grouping, Classification

ACM Reference Format:

Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3184407.3184412>

1 INTRODUCTION

The increasing complexity of software systems requires tools and techniques for monitoring the behavior of large and complex applications. Many of these tools trace an application by recording events at run time and writing them to a trace file for later analysis. For example, a memory monitoring tool could record object allocations and garbage collector activity (e.g., object moves) so that the application's heap can be later reconstructed offline for various analyses.

Such monitoring tools produce huge amounts of data, which have to be classified, grouped and visualized in order to be helpful

for the user. For example, users might want to know how many objects of a certain type were allocated, at which locations they were allocated, and how long they survived. Unfortunately, many state-of-the-art tools fail to provide a flexible information retrieval technique. Most of them only support hard-coded classification criteria (often *type* is the only one) in conjunction with tabular histograms, e.g., showing the number of instances per class and the number of allocated bytes. They don't allow users to classify the data based on multiple criteria (e.g., type, allocation site and age) and miss features to organize and aggregate the resulting information hierarchically on multiple levels.

Our tool AntTracks [12, 13] is a memory monitoring tool for Java based on the Java Hotspot™ VM [21] that records object allocations and garbage collection moves. It also offers offline analysis of trace files, in which the heap can be reconstructed for any garbage collection point in time. Bitto et al. [3] showed how to reconstruct an application's heap from traces produced by AntTracks. Based on this work, Weninger et al. [25] presented first ideas on object classifiers with the goal to make the classification of memory monitoring data more general and customizable.

In this paper, we extend our work by presenting a generally applicable object classification and multi-level grouping concept. An object classifier processes an object and classifies it based on a certain criterion derived from the object's properties, e.g., classifying heap objects based on their type. Objects with the same classification result are grouped together. As already mentioned, most state-of-the-art memory monitoring tools have two major restrictions: (1) They only offer a restricted set of classification criteria, such as *Type* or *Allocation Site*, and (2) their grouping mechanism is based on just a single classification criterion, i.e., single-level grouping. Our approach eliminates both restrictions. In addition to a set of predefined object classifiers that are usable out-of-the-box, users can define custom object classifiers as small dynamically-loaded code snippets. Furthermore, the grouping is not based on a single criterion but on dynamic classification trees, i.e., on multi-level grouping based on multiple object classifiers. Such classification trees store classification results in a hierarchical manner and allow a more flexible top-down data analysis approach. The concepts of object classification, multi-level grouping and classification trees are not restricted to memory data and may therefore also be used in other domains.

Our scientific contributions are (1) a novel concept of *object classifiers*, a way to classify a collection of objects based on their properties, (2) a multi-level grouping algorithm that classifies a collection of objects based on a user-chosen set of object classifiers into a *classification tree*, (3) various classification tree data structures

ICPE '18, April 9–13, 2018, Berlin, Germany

© 2018 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*, <https://doi.org/10.1145/3184407.3184412>.

that differ in terms of classification throughput, memory overhead and information loss, and (4) a quantitative evaluation based on well-known benchmarks as well as a functional evaluation based on typical memory analysis use cases.

2 BACKGROUND

AntTracks consists of a virtual machine based on the Java Hotspot™ VM and a memory analysis tool. The AntTracks VM records memory events into trace files, which can then be analyzed offline with the tool. Since our object classifier approach has been integrated into this tool, it is essential to understand AntTracks’s architecture and workflow.

2.1 Trace Recording

The AntTracks VM records memory events, e.g., events for object allocations and object movements executed by the garbage collector (GC), throughout an application’s execution and writes them into trace files. Furthermore, it is also capable of recording pointers between objects [11]. After loading such a trace file, the AntTracks analysis tool provides overview of the memory behavior over time and can reconstruct the heap’s state and layout for every garbage collection point by incrementally processing the events in the trace.

2.2 Trace Reconstruction and Data Structure

Bitto et al. [3] show that a naïve approach, in which every heap object is represented by a Java object in the analysis tool, would result in an unacceptable memory overhead. Therefore, we developed the data structure shown in Figure 1. It separates the heap into multiple spaces. For example, the *ParallelOldGC*’s heap consists of one eden space, two survivor spaces, and one old space. Each of these spaces encompasses various fields such as the starting address, the size, or the kind of the space (i.e., *eden*, *survivor* or *old*). Additionally, each space contains an address-to-LAB map. A LAB (local allocation buffer) represents a sequence of objects that have been processed together by the same thread (e.g., objects that have been allocated by the same thread within the same thread-local allocation buffer (TLAB)). Each entry in the LAB’s object array represents one heap object and contains a pointer to a global cache of object representations, called *ObjectInfo*. *ObjectInfos* are cached structures that contain information which is shared by multiple objects, namely the event which created the object (e.g., an allocation by the interpreter), the object’s allocation site, its type and its size. For array allocations, also the array length is stored. Using this mechanism, many different objects can be represented by the same *ObjectInfo*. Their addresses do not have to be explicitly stored

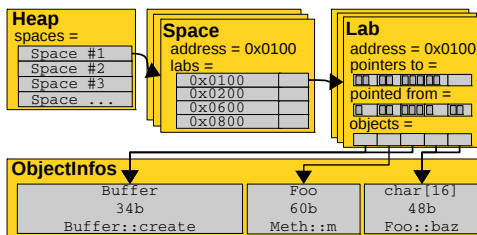


Figure 1: AntTracks’s data structure to represent a heap at a certain point in time.

but can be computed from their LAB’s address. In addition to the object array, each LAB contains two arrays of the same length to store pointer information. For each entry in the object array, i.e., for each heap object, the respective entry in the *pointers to* array contains the addresses of all objects that are referenced by this object. Analogously, each entry in the *pointed from* array contains the addresses of all objects that point to the respective object.

3 APPROACH

This section presents the domain-independent concepts of classification (i.e., representing an object by a classification result made up of one or more classification values) and multi-level grouping (i.e., arranging classification results in a tree structure). Examples on how these concepts can be applied in a specific domain / tool will be given in the context of Java and the classification of Java heap objects within the AntTracks memory analysis tool. If a specific heap state is shown, it has been reconstructed from a trace of a *DaCapo xalan* benchmark run.

3.1 Source Collection and Source Objects

Classification and grouping always operate on a *source collection* which consists of *source objects* of a certain type. AntTracks’s source collection when classifying a heap state are the Java heap objects that have been live at the given point in time.

The source collection does not have to be represented by a single class but may be made up of multiple classes that interact with each other, see Figure 2. One of these classes must act as *the* source collection to the public. This class is required to provide functionality to iterate the contained source objects. In AntTracks, as explained in Section 2, a heap state is modeled by multiple classes (i.e., the heap itself, which further consists of multiple spaces, which further consist of multiple LABs), yet the Heap class acts as the source collection to the public.

Similarly, the properties of a source object do not have to be stored in a single object. In AntTracks, for example, they are stored in different locations: Most of them are stored in the *ObjectInfo*, but a heap object’s pointers are stored in the LAB, and its address is calculated on demand.

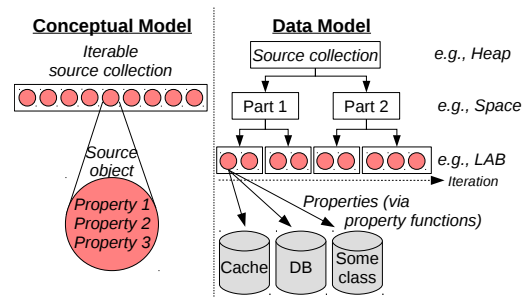


Figure 2: Basic classification concepts: Source collection, source objects and source object properties.

We distinguish the term *object* from the term *source object* because *object* is often used in the context of programming languages to describe a certain instance of a class. A source object, on the other hand, represents properties that may be stored in various places.

3.2 Source Object Properties and Source Collection Iteration

A source object is described by its m properties based on its position P within its source collection, as shown in Definition 3.1.

Definition 3.1. A source object at position P within its source collection is described by its m distributed properties:

$$so_P \text{ is described by } (prop_1, prop_2, \dots, prop_m)_P$$

P 's format depends on the source collection. For example, in a list, source objects are identified by their index i , i.e., $P = i$. In AntTracks's heap data structure, a source object's position, i.e., the position of a heap object within the reconstructed heap, is described by (1) the space in which the object is, (2) the lab inside the object's space, and (3) the object's position within the lab, i.e., $P = (spaceIndex, labIndex, objectIndex)$.

Source collection iteration describes the task of visiting every position in the source collection and obtaining the properties of the respective source object. In AntTracks, iterating the heap means to visit every element in the ObjectInfo array of every LAB in every space, and collecting all properties of the currently visited heap object, e.g., calculating its address based on its containing LAB.

3.3 Object Classifiers

As soon as a source object's properties have been obtained, object classifiers can be used to classify it. Object classifiers are entities that classify a source object based on a certain criterion derived from the source object's properties. Each object classifier provides a `classify` function, which takes one parameter per source object property and returns the classification result. Additionally, every object classifier contains the following meta-data:

Name. A unique name used to identify the classifier.

Return Type. The `classify` method's return type.

Description (Optional). Useful to keep the classifier's names short while still offering additional information about the classifier's purpose.

Example (Optional). A possible classification result returned by the classifier, e.g., `java.lang.Integer` returned by AntTracks's *Type classifier*. This can be shown as a classification sample to the user in the UI.

Cardinality. Each classifier can be of one of the following three cardinalities: *One-to-one*, *one-to-many* or *one-to-hierarchy*. Depending on the cardinality, the classifier's classification result may be made up of a different number of classification values, see Figure 3.

In AntTracks, object classifiers are used to classify Java heap objects based on their properties such as the object's type, its allocation site and so on. Each classifier, e.g., the *Type classifier*, implements a common Java interface (most importantly the `classify` method), see Section 4.2.

3.3.1 One-to-one Classifier. A *one-to-one classifier* classifies a source object by a unique classification value as classification result (see top part in Figure 3). The returned classification value is an instance of the classifier's return type, i.e., a one-to-one String classifier returns a single String as value.

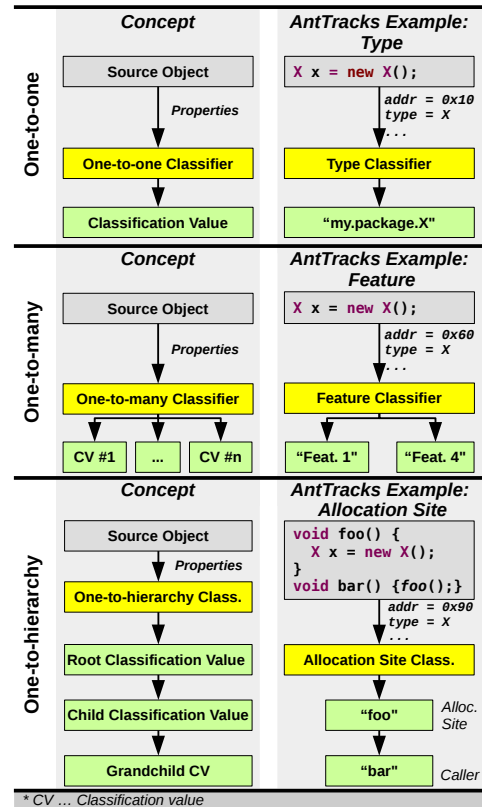


Figure 3: Object classifiers classify a source object based on its properties. The three types of classifiers vary in their classification value cardinality.

An example for a one-to-one classifier is AntTracks's predefined *Type classifier*, which classifies a Java heap object based on its type's name. Figure 4 shows a part of AntTracks's analysis view where each heap object has been classified using the *Type classifier*. Overall

Name	Objects		Bytes	
	#	%	#	%
Overall	11.077.848	100,0	1.061.744.056	100,0
java.nio.HeapCharBuffer	3.919.909	35,4	188.155.632	17,7
char[]	3.886.043	35,1	237.942.968	22,4
java.util.HashMap\$Node	736.420	6,6	23.565.440	2,2

Figure 4: Classifying heap objects by type in AntTracks.

shows the number and byte count of the whole heap, and each child row represent a group of heap objects that have been classified by the same value, i.e., that are of the same type. Each heap object is part of exactly one group, i.e., *one-to-one* classification.

Filters. Filters are a special kind of *one-to-one classifiers*, which are of type Boolean. Filters are used in the classification process to define whether a source object should be further processed by subsequent operations.

3.3.2 One-to-many Classifier. A *one-to-many classifier* classifies a source object by multiple classification values, as can be seen in the middle part of Figure 3. The result is a set of instances of the classifier's type: If the classifier's type is String, a set of strings will be returned.

An example for such a classifier is the predefined *Feature classifier* in AntTracks. Assume that (possibly overlapping) code ranges represent specific features [10]. The allocation site of an object may then belong to one or more of these features. The *Feature classifier* performs a feature mapping for every Java heap object and returns the set of features to which its allocation site belongs. Figure 5,

Name	Objects		Bytes	
	#	%	#	%
Overall	11.077.848	100,0	1.061.744.056	100,0
java	9.025.314	81,5	478.658.200	45,1
xml	1.604.657	14,5	542.529.088	51,1
others	191.918	1,7	27.304.992	2,6

Figure 5: Classifying heap objects by feature in AntTracks.

similar to Figure 4, shows again a part of AntTracks’s analysis view. This time, each heap object has been classified using the *Feature classifier*. Since the *Feature classifier* is a *one-to-many classifier*, each heap object can be part of multiple groups (if the classifier returned multiple values, i.e., features, for that heap object).

3.3.3 *One-to-hierarchy Classifier*. A *one-to-hierarchy classifier* classifies a source object by hierarchical classification values, as shown in the bottom part of Figure 3. Such a classifier returns objects of the classifier’s return type in an ordered list. The object at index 0 is the root object, and for all $i > 0$ the object at index $i - 1$ is the parent of the object at index i .

An example for a *one-to-hierarchy classifier* is the predefined *Allocation Site classifier* in AntTracks, which classifies an object based on its allocation site and the allocation’s call sites. The root object (at index 0) is the code location where the object was allocated, the object at index 1 is the code location from where the allocating method was called, and so on (i.e., the code location at index i is the callee and the code location at index $i + 1$ the caller). Figure 6

Name	Objects		Bytes	
	#	%	#	%
Overall	11.077.848	100,0	1.061.744.056	100,0
java.nio.CharBuffer.wrap(char[], int, int) : java.nio.Cha...	3.919.909	35,4	188.155.632	17,7
sun.nio.cs.StreamEncoder.implWrite(char[], int, int)...	3.916.234	35,4	187.979.232	17,7
sun.nio.cs.StreamEncoder.write(char[], int, int) : ...	3.916.234	35,4	187.979.232	17,7
sun.nio.cs.StreamDecoder.implRead(char[], int, int)...	3.675,0	0,0	176.400,0	0,0
sun.nio.cs.StreamEncoder.write(java.lang.String, int, i...	1.725.368	15,6	44.568.696	4,2
java.io.OutputStreamWriter.write(java.lang.String, i...	1.725.368	15,6	44.568.696	4,2
java.io.Writer.write(java.lang.String) : void : 7	1.725.368	15,6	44.568.696	4,2

Figure 6: Classifying heap objects by allocation site in AntTracks.

also shows a part of AntTracks’s analysis view similar to Figure 5, yet each heap object has been classified using the *Allocation Site classifier* instead. First-level children of the *Overall* group, i.e., row 2 and row 6, are allocation sites where objects have been allocated. Child relations represent the call chain, e.g., the call sites on row 3 and row 5 called the allocation site on row 2, and the call site on row 4 has been the single caller to the call site on row 3.

3.4 Multi-level Grouping

Single-level grouping splits a set of objects into multiple groups. Each group represents a distinct classification result (i.e., the classifier’s return value) and contains all objects that are classified by

this result. Typical single-level grouping only supports one-to-one classifiers, i.e., each object is mapped to exactly one classification value. In addition to introducing other classifier types beside one-to-one classifiers, we present multi-level grouping to enhance the flexibility and level of analysis detail.

3.4.1 *Classification*. Similar to single-level grouping, multi-level grouping is an operation that groups a set of source objects. Yet, instead of applying a single classifier, a list of classifiers is applied one after the other to every source object, and the sorted list of their classification results (where each classification result may be made up of multiple classification values) make up the source object’s classification.

Obj.	Classification and results in parentheses
O(1)	[Age(1) → Feat(F1, F2) → AS(add, A)]
O(2)	[Age(1) → Feat(F1, F2) → AS(add, B, D)]
O(3)	[Age(3) → Feat(F1) → AS(main, C, A)]
O(4)	[Age(3) → Feat(F1) → AS(clone, D)]
O(5)	[Age(3) → Feat(F1) → AS(main, C, A)]
O(6)	[Age(1) → Feat(F1, F2) → AS(add, A)]

Table 1: Example classification of 6 Java heap objects based on three classifiers: Age (one-to-one), feature (one-to-many) and allocation site (one-to-hierarchy).

Table 1 shows an example classification for six objects $O(1)$ to $O(6)$. The three classifiers that get applied are (1) the *Age classifier*, a one-to-one classifier categorizing heap objects based on their number of survived GCs, (2) the *Feature classifier* (see Section 3.3.2) and (3) the *Allocation Site classifier* (see Section 3.3.3). Each classification contains three classification results, one per classifier, sorted in the order in which the classifiers were applied.

3.4.2 *Classification Tree*. Raw information as presented in Table 1 is not very helpful for the user. Classification trees bring such classification results into a hierarchical format that allows (1) flexible processing of data, such as merging, subgrouping, counting and so on as well as (2) straightforward visualization, e.g., as a tree table view, for user-driven analysis.

Figure 7 shows the creation of a classification tree for the objects in Table 1. Rectangles (yellow) represent tree nodes containing their keys as text, and arrows point to their child nodes. Smoothed rectangles (blue) represent the data that a node is holding, i.e., the source objects assigned to the node.

The following example explains how $O(1)$ gets added to the classification tree. The algorithm starts with the root node as the *current node*. During the classification process, when looking for a child node with a certain key that does not exist yet, a new child gets created for that key.

The *Age classifier* returns 1 as the classification result for $O(1)$. For each current node (i.e., the root node), the child matching this classification becomes the new current node, i.e. the status of current node moves from the parent to the child. Then, the *Feature classifier* is applied, which returns $F1$ and $F2$ as its classification values for the source object $O(1)$. Both features get added as children of 1 and become the new current nodes. Finally, the *Allocation Site classifier* gets applied on the source object and returns the allocation site *add* and its caller *A*. *add* nodes are appended as children

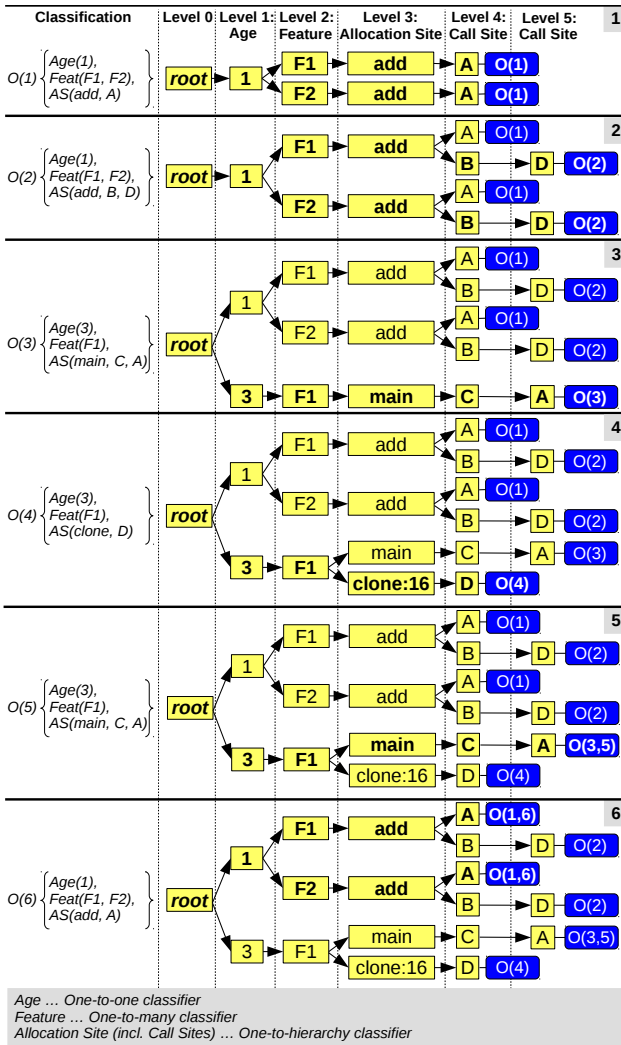


Figure 7: Step-by-step multi-level grouping of six heap objects into a classification tree based on age, feature and allocation site.

to all current nodes (i.e., to *F1* and *F2*) and *A* nodes are appended to the two *add* nodes.

Since no more classifiers have to be applied, the object is then added as a data entry at the current nodes, i.e., at both *A* nodes. This is the state that is shown in the top part of Figure 7. To reach the state at the bottom of Figure 7 the above steps are repeated for every source object *O(2)* to *O(6)*.

Figure 8 shows an example on how classification trees get visualized in AntTracks. It displays a part of AntTracks’s heap state analysis view where all heap objects have first been classified by *Age*, then by *Feature*, followed by *Allocation Site*.

3.5 Data Representation in Nodes

Source objects have to be associated with certain nodes of the classification tree. Various approaches are possible, some of which sacrifice information in favor of reduced memory overhead (see Figure 9).

Name	Objects		Bytes	
	#	%	#	%
Overall	11,077,848	100,0	1,061,744,056	100,0
9	1,109,019	10,0	106,448,816	10,0
java	903,605	8,2	48,617,760	4,6
xml	160,560	1,4	54,717,928	5,2
org.apache.xpath.axes.LocPathIterator.asIterator(...)	32,177	0,3	1,287,080	0,1
org.apache.xpath.axes.NodeSequence.cloneWith...	12,369	0,1	494,760	0,0
org.apache.xpath.objects.XNodeSet.getFresh...	6,828	0,1	273,120	0,0
org.apache.xpath.objects.XNodeSet.iterator() : org...	5,541	0,1	221,640	0,0
org.apache.xml.dtm.ref.ExpandedNameTable.init...	7,685	0,1	245,920	0,0

Figure 8: AntTracks’s visualization of classification trees.

3.5.1 Lossless Approaches. Information lossless approaches allow to retrieve all properties of all source objects stored in the classification tree. This is needed if the classification tree should later be used for further complex processing.

Naïve List Approach. A naïve approach is to represent the node’s data as a list of objects. A source object’s properties (which are distributively stored) would have to be combined into a new object on demand (e.g., new `MyObject(p1, p2, p3)`).

We chose to store source object properties in a scattered way exactly because we want to *prevent* the creation of class instances, which would lead to increased memory footprint (e.g., due to object headers). Further, the more live objects reside in the heap, the less memory is available for new allocations. This results in more frequent GC invocations, which may slow down the application.

Property List Approach. Instead of storing a list of objects, this approach only stores a list of one of the source object’s properties. This is possible if the object’s remaining properties can be derived from this property, which is the case for nearly all use cases. In AntTracks, for example, heap objects can be identified by their address. The downside of this approach is the additional indirection when obtaining the other properties on demand.

3.5.2 Lossy Approaches. convey a feeling The lossless approaches retain object identity, i.e., we know exactly which source objects have been added to which tree nodes. This level of detail may be traded for less memory-consuming tree node data structures.

Mapping Approach. This approach relies on a map, where the key’s type is application-dependent and the value is represented by a counter.

When adding a source object to a node, information of interest about the object gets extracted as the *object key*. This object key is

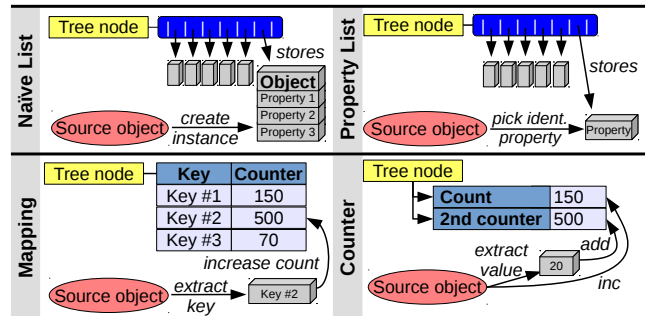


Figure 9: Two lossless list approaches and two lossy approaches based on counters to store node data.

then looked up in the node’s data map and the respective counter gets incremented (or created if it does not yet exist).

It is crucial to take two aspects into account when choosing the object key: (1) What data should be reconstructed from the classification tree and (2) that many source objects should share the same object key to keep the number of entries in the map small. For example, AntTracks uses the object size (in bytes) as the object key. While this allows to only aggregate the number of objects and number of bytes represented by a certain node, it offers high memory saving potential which is discussed in more detail in Section 5. For example, if 1000 objects of only three different sizes get added to the same node, this approach just needs three key-value pairs compared to 1000 list entries as in the list approaches.

Counter Approach. This approach is designed to have the lowest memory footprint, while giving up flexibility and accepting the highest loss of information. Every time a source object gets classified at a certain node, counters stored in the node get incremented based on a fixed scheme. In AntTracks, for example, we could store two counters, one for the number of objects and one for the number of bytes classified at the given node.

This approach even loses information about specific properties. For example, it would not be possible to determine how many heap objects of a certain size have been classified, which is possible using the mapping approach.

3.6 Aggregation and Duplicate Detection

Using a one-to-many classifier may cause a source object to be added to multiple nodes. To avoid wrong results when aggregating this data, we have to detect duplicate entries in the tree and ignore them.

3.6.1 List Approaches. Since the entries in every data list are distinct, the lists can be treated as sets. The set of objects in a tree with head n can be computed recursively as the union of the objects in n and in the subtrees (Equation 1). Duplicates will be removed and the resulting set can be used for counting.

$$objects(n) = n.data \cup \left(\bigcup_{n.children}^{child} objects(child) \right) \quad (1)$$

3.6.2 Mapping Approach. By extracting a source object’s object key, we lose the object identity which would be needed for duplicate detection. Therefore, we additionally have to keep track of multiple classifications. This can be done by installing a second map, i.e., the *duplicate map*, in each node.

If a source object is added to more than one subtree of a node n , a counter for the object’s key is incremented in the duplication map of node n , which is later used for sifting out duplicates when the total number of objects in a tree is computed.

3.6.3 Counter Approach. Similar to the mapping approach, every node could store a duplicate counter per data counter. In all situations where a duplicate counter in the mapping approach would be incremented, the duplicate counter in the counter approach is incremented.

3.7 Advanced Classifiers

For advanced use, a special kind of classifiers are *transformers*. So far, a classifier always took a source object’s properties as its input and returned one or more classification values as classification result. A transformer takes a source object and (1) transforms it

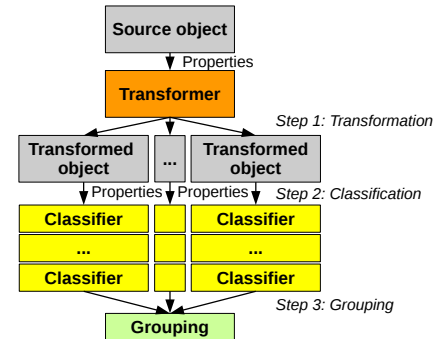


Figure 10: Transformers transform a source object into a set of other source objects, classify each of these objects and group them.

into a set of other source objects, (2) classifies each of these objects based on a selected set of object classifiers, and (3) multi-groups them based on their classification results (see Figure 10).

A use case for transformers in the domain of memory monitoring is pointer analysis. First, a heap object gets transformed into the set of all objects that are referenced by it. Second, this set of objects gets classified based on a list of other classifiers selected by the user. Finally, the classification results get multi-grouped into the resulting classification tree. For example, this can be used to analyze type-points-to-type graphs, as done by Jump and McKinley [8, 9].

4 IMPLEMENTATION

The previous section explained the domain-independent core concepts of classification and multi-level grouping based on object classifiers alongside some examples in the context of AntTracks. This section discusses some implementation details on how these concepts have been incorporated into AntTracks and its memory analysis.

Property	Additional info
address	
space	Space index, name, address, length, ...
type	Name, package, fields, ...
size	The object’s size in bytes
isArray	true / false
arrayLength	-1 for non-arrays
allocationSite	Call stack, ...
pointedFrom	Addresses of all referencing objects
pointsTo	Addresses of all pointees
eventType	Allocation event (alloc. subsystem, ...)

Table 2: Source object properties for heap objects.

4.1 Source Objects: Java Heap Objects

AntTracks’s source objects are Java heap objects that were alive in the monitored application at a given point in time, i.e., the heap

objects that make up a certain heap state. Table 2 shows which properties make up a Java heap object in AntTracks, i.e., the source object properties. Every object classifier classifies a Java heap object based on a criterion derived from these properties.

4.2 Object Classifiers

In AntTracks, classifiers implement a common base interface. This interface defines the `classify` method, with its parameter signature matching the Java heap object properties.

To provide a convenient analysis environment for most use cases, AntTracks comprises multiple predefined object classifiers. These classifiers, listed in Table 3, can be used and combined freely on every heap state. An example implementation of the *Type classifier* can be seen in Listing 1.

Listing 1: Implementation of the *Type classifier* in AntTracks.

```
public class TypeClassifier implements Classifier<String> {
    // ... Fields modifiable by user, e.g., showPackage ...
    @Override public String classify(
        long address, Space space, Type type, long size,
        boolean isArray, int arrayLength, AllocationSite allocSite,
        long[] pointedFrom, long[] pointsTo, Event eventType) {
        return type.getName(showPackage);
    }
}
```

When a heap state is opened in AntTracks, a default classification (*Type classifier* followed by the *Allocation Site classifier*) gets applied. This gives a fast overview that shows which types have the most living objects, and where these objects have been allocated.

4.3 Heap Iteration

We implemented three different iteration approaches for AntTracks’s heap data structure to evaluate their influence on the classification speed.

4.3.1 Java Streams. This approach has been implemented as a baseline for performance comparison. It uses the default technique for Java streams on custom data structures by implementing a *SplitIterator*, the concurrent counterpart of an *Iterator*.

Java Stream Memory Overhead. The main problem with Java streams and spliterators is that they are generic classes working on Java objects of type `T`. Therefore, to support Java streams in AntTracks, we have to transform AntTracks’s source objects (i.e., heap objects that are stored as scattered properties) into instances of an auxiliary `HeapObject` class. These short-living objects (which only exist while the stream is processed) may put unnecessary burden on the garbage collector, especially for large heap states.

4.3.2 Fake SplitIterator. This approach relies on a custom iteration class that provides a `tryAdvance` and a `trySplit` method, similar to the *SplitIterator* implemented for the Java stream approach. However, this *fake spliterator* does not inherit from Java’s *SplitIterator* interface, but only mimics its behavior. More specifically, the *fake spliterator*’s `tryAdvance` does not match the official interface but has been changed in a way that allows the *fake spliterator* to process a heap object’s properties separately, which has the advantage of avoiding the need for auxiliary objects.

4.3.3 Integrated Iteration Functions. A basic implementation of this approach already existed in the previous versions of AntTracks. It provided sequential iteration functions on each data structure level, i.e., on the `Heap`, the `Space`, and the `LAB`. In our approach, we added support for parallel iteration, which significantly increased performance.

4.4 User-defined Classifiers

Classification in AntTracks is not restricted to predefined classifiers, but allows users to define new classifiers, i.e., *user-defined classifiers*, in two different ways: (1) By using Java’s Service Provider Interface (SPI) concept, where new classifiers can be added to AntTracks as pre-compiled JAR files, and (2) by using in-memory on-the-fly compilation to support classifier development at run time.

4.4.1 Service Provider Interfaces (SPI). A *service provider interface* is a set of public interfaces and abstract classes that a third-party developer can implement. In AntTracks, the SPI encompasses abstract classes for classifiers, transformers, and filters. All of them define an abstract `classify` method which can be implemented by third-party developers in a sub-class. If a JAR containing such an implementation is detected on AntTracks’s class path (using convenient SPI methods), it will be added to the list of available classifiers or filters.

4.4.2 On-the-fly Compilation. It is also possible to define new object classifiers, transformers and filters at run time. For example, whenever users have to select one of the available classifiers, they are offered to define a new one. The user then has to provide the `classify` method, the classifier’s name, description, example and cardinality. This information gets merged into an object classifier template file which will then be compiled with a modified Java compiler that enables compilation without generating a Java class file on disk, i.e., the classifier gets compiled in-memory and on-the-fly.

This compilation relies on the `JavaCompiler` instance returned by `ToolProvider.getSystemJavaCompiler()`. This instance allows modifying the compilation process in various ways. The most important step is to provide a modified `JavaFileManager`. Instead of providing a stream to a file on disk, AntTracks’s version returns a `ByteArrayOutputStream` that keeps a class’s byte code stored in memory. Additionally, the file manager’s class loader has been modified to not only look up classes stored on disk, but also to look up classes that are stored in memory.

5 EVALUATION

To evaluate the applicability of AntTracks’s object classifiers and multi-level grouping we show how one can use the tool to detect memory leaks and how to reproduce memory classification done in related work.

Even though lossless classification tree implementations may be needed in certain situations, a lossy approach provides enough information for most use cases, including AntTracks’s heap state analysis. Therefore, another goal of this evaluation is to analyze how much classification throughput can be gained as well as how much memory can be saved by accepting the information loss due to using a lossy classification tree implementation. All of these

Name	Description
Address	Classifies objects based on their address.
Type	Classifies objects based on their type's name.
Allocating Subsystem	Either <i>VM</i> , <i>Interpreter</i> , <i>C1-compiled code</i> or <i>C2-compiled code</i> .
Array Length	Classifies array objects based on their length. Non-array objects are classified as -1.
Object Kind	Either <i>Instance</i> (class instances), <i>Small Array</i> (< 255 elements), or <i>Big Array</i> (≥ 255 elements).
Space	Classifies objects based on the heap space in which they are contained.
Space Mode	Classifies objects based on the mode, i.e., a GC-dependent space info, of their containing heap space.
Space Type	Classifies objects based on the type (e.g., <i>Eden</i>) of the space in which they are contained.
Feature	Classifies objects based on a loaded feature-to-code mapping file.
Allocation Site	Classifies objects based on their allocation site (allocating method + var. number of call sites).
Pointed From	This transformer is used to classify the objects that reference a given object.
Points To	This transformer is used to classify the objects that a given object references.

Table 3: Predefined classifiers in AntTracks.

analyses have been conducted based on well-known benchmarks using three different classifier combinations: (1) Type classifier (2) Allocation Site classifier (3) Type classifier, followed by the Allocation Site classifier.

Setup. All measurements were run on an Intel® Core™ i7-4790K CPU @ 4.00GHz x 4 (8 Threads) on 64-bit with 32 GB RAM and a Samsung SSD 850, running Ubuntu 17.10 with the Kernel Linux 4.13.0-16-generic. All unnecessary services were disabled in order not to distort the experiments.

5.1 Performance Evaluation

The goal of this evaluation is to gain insight into how much the classification throughput increases when giving up object identity and if Java streams are suitable to iterate distributed source objects. Thus, we compare both implemented tree node types (i.e., the property list approach (lossless) and the mapping approach (lossy)) using three different parallel heap iteration techniques (i.e., Java stream, fake spliterator and integrated iteration).

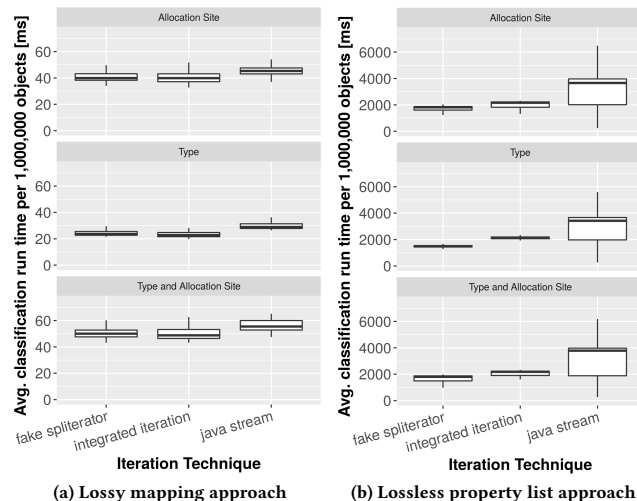


Figure 11: Performance comparison between the mapping approach and the property list approach.

We used the DaCapo [4] and the DaCapo Scala [6] benchmark suites, in which, according to Lengauer et al. [14], h2 and factorie are the benchmarks with the largest live set. We chose to only analyze these two benchmarks since the other benchmarks from the

mentioned suites do not provide heap states in the same dimension. Both trace files (h2: 2.9 GB trace file covering 26 garbage collections with 15,800,000 objects on average per heap state; factorie: 19.5 GB trace file covering 205 GCs with 8,600,000 objects on average per heap state) have been parsed and a classification tree has been generated at every garbage collection end using every parameter combination (i.e., iteration type, classifier, tree type).

Figure 11a shows the average throughput of this classification tree generation when using the lossy mapping approach, while Figure 11b shows the throughput using the property list approach. We can see that the mapping approach is orders of magnitude faster than the property list approach due to the work that is needed to add the object's address to the sorted data list when using the property list approach. This strengthens our assumption to use the mapping approach when object-identity loss is acceptable.

Furthermore, it shows that heap iteration using Java streams is in general slower than the other two approaches. Especially for larger heap states, the streaming approach falls behind the other approaches. As hypothesized, this may be due to the temporary objects that have to be generated during the iteration. Independent of the domain this indicates that Java streams are not suitable for iterating distributively stored source objects. The fake spliterator approach is able to scale and parallelize the best, which explains its advantage when using the property list approach.

5.2 Memory Footprint

Beside providing the better classification performance, it is interesting to see how much memory can be saved when using the object-identity-losing mapping approach instead of the property list approach.

We analyzed a traced run of every DaCapo and DaCapo Scala benchmark and reconstructed the heap state after every garbage collection, if the heap state contained at least 200,000 objects. The *Type classifier* showed that the number of types of live objects at a certain point in time is approximately the same across all benchmarks (around 500 objects), independent of the number of live objects. Some of the benchmarks have few live objects with a high number of different allocation site nodes (i.e., few objects allocated at different sites) while some benchmarks with a large number of live objects only generate a small number of allocation site nodes (i.e., a lot of objects allocated at the same sites). Nevertheless, the tree never reached a critical size in terms of node count for any of the tested applications (tree size always below 20,000 nodes).

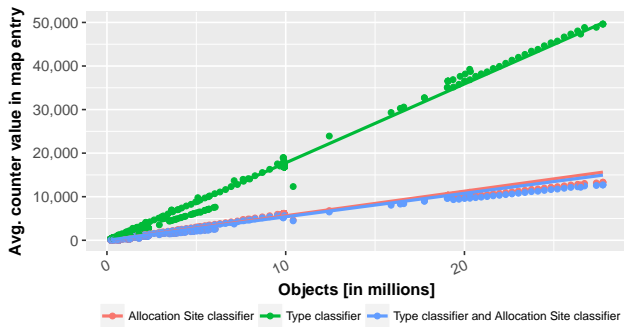


Figure 12: Average object count per data map entry using the mapping approach.

Figure 12 shows that with a rising number of classified objects, the average number of objects represented by a single data map entry in the mapping approach increases. For example, classifying about 10,000,000 objects based on the *Type classifier* resulted in data map entries each representing about 18,000 objects on average (see regression line in Figure 12). Assume that the property list approach is implemented using arrays and needs 8 bytes per classified object (i.e., the heap object’s 64-bit address excluding memory needed by auxiliary data structures). Let’s further assume that each map entry in the mapping approach points to a key (containing an int) and a value (containing a long), thus taking up $3 \cdot 16$ ($3 \cdot \text{VM header}$) + $2 \cdot 8$ ($2 \cdot \text{pointer}$) + 4 (int) + 8 (long) = 76 bytes. If one such data map entry represents 18,000 objects, the property list approach ($8 \cdot 18,000$ bytes) consumes about 1900 times as much memory as the mapping approach (76 bytes).

Based on these results and those presented in Section 5.1, we decided to use classification tree generation based on fake spliterator heap iteration and the mapping approach in AntTracks.

The next section shows that the lossy mapping approach still provides enough information to detect memory leaks and allows general memory analysis.

5.3 Functional Evaluation

AntTracks’s goal is to provide a general memory monitoring and analysis tool that primarily focuses on developers and their needs, for example performing memory leak detection. In addition, user-defined classifiers, their flexible combination, and multi-level grouping allows developers and also researchers to use AntTracks for more general and experimental memory analyses.

5.3.1 Memory Leak Detection. Memory leak detection is the main task developers perform when using AntTracks. To evaluate AntTracks’s ability to allow memory leak detection, as well as finding the root cause, we used it on an example artificial application that uses a stack¹ for storing its data. It first pushes 1 million objects onto the stack, then pops these 1 million objects, followed by another 100,000 pushes and another 100,000 pop operations. Opening the application’s trace displays the overview shown in Figure 13. We can clearly see that we miss a drop of the number of live objects after the 1 million objects got popped from

¹<https://www.codeproject.com/Articles/30593/Effective-Java; Item 6: Eliminate obsolete object references; last accessed October 17, 2017>

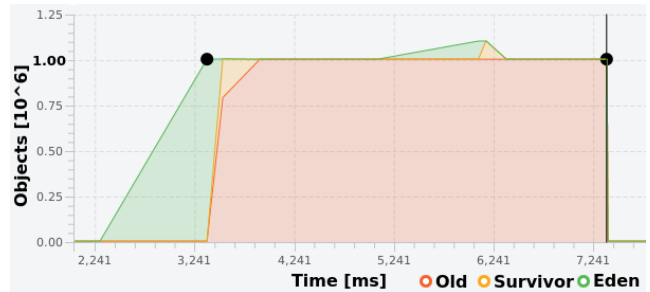


Figure 13: Object count overview of the buggy stack implementation.

the stack, as we would expect in a non-faulty implementation. To further investigate this problem, we utilized AntTracks’s heap diffing functionality, which also supports object classifiers and allows to analyze heap changes over time. Figure 14 shows the application

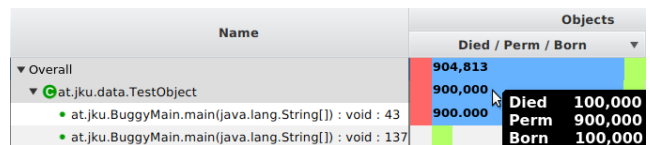


Figure 14: Heap diff of the buggy stack implementation.

of the *Type classifier* followed by the *Allocation Site classifier* on the time frame selected in Figure 13 (black dots). On the type node (2nd row, `at.jku.data.TestObject`), we can see that only 100,000 objects of this type were deallocated (red bar), while exactly the same amount of objects were allocated (green bar). 900,000 objects stayed alive during the whole time frame (blue bar). Looking at the indented allocation site nodes (3rd and 4th row), we see how many `TestObject`s that were originally allocated at these sites were born, have survived, or have died.

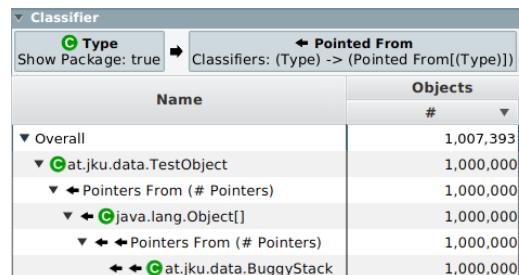


Figure 15: Pointer analysis of the buggy stack implementation.

Additionally, we would like to know which objects keep those objects alive. Figure 15 shows a rather advanced application of object classifiers: It first classifies a given object by its type, then transforms that object into its set of referencing objects, classifies them by type and then transforms them again into their sets of referencing objects, finally classifying those objects by type. It shows that the `TestObject` instances are referenced from the type `Object[]`, which is again referenced by the type `BuggyStack`. With this information, it is easy to find the bug in the source code. `BuggyStack` is a faulty stack implementation that keeps references to previously

stored objects even after pop operations until a subsequent push operation overwrites them.

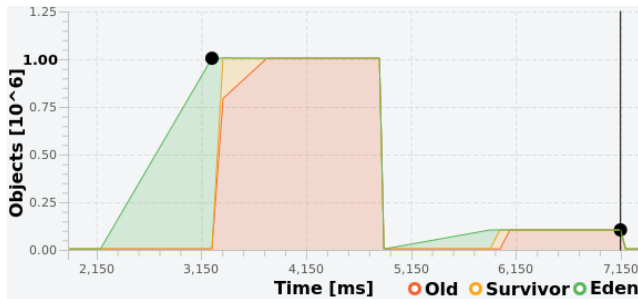


Figure 16: Object count overview of the fixed application.

Name	Objects		
	Died	Perm	Born
Overall	1,000,147		
at.jku.data.TestObject	1,000,000		
at.jku.BuggyMain.main(java.lang.String[]): void : 137		Died: 1,000,000	
at.jku.BuggyMain.main(java.lang.String[]): void : 43		Perm: 0	
	1,000,000	Born: 100,000	

Figure 17: Heap diff of the fixed application.

Figure 16 and Figure 17 show the object counts and the heap diff results after the stack implementation was fixed.

5.3.2 Memory Analysis. Developers as well as researchers may want to classify heap objects based on a criterion not yet covered by one of the predefined classifiers, which is possible by writing a *user-defined* classifier. To showcase the implementation of *user-defined* classifiers, we searched for related work on heap object classification. For example, Mitchell and Sevitsky [17] classified heap objects in terms of *collection health* and *instance health*. Both classification criteria have been successfully implemented as *user-defined* classifiers and can be used and freely combined with other classifiers in AntTracks.

Collection Health. Collection health classifies every heap object as one of four types, depending on its use inside collections. (1) *head*, the head of a collection, e.g., `HashMap`, (2) *array*, array backbones, e.g., `HashMap$Entry[]`, (3) *entry*, recursive list-style elements, e.g., `HashMap$Entry`, and (4) *contained*, anything else.

The classification for collection health is a typical use-case for a user-defined one-to-one object classifier. Every object gets classified by exactly one value, i.e., either *head*, *array*, *entry* or *contained*. According to Mitchell and Sevitsky, every object that is an array of a reference type gets classified as *array*. This is straightforward to check in the classifier implementation² since we know the object’s type. If an object is not classified as *array*, it falls in the *entry* category if it is of a type `T` and references an object of the same type `T`. This check can be accomplished by following and analyzing the pointers in the object’s pointer array. If the object has not been categorized as *array* or *entry*, the object’s pointers are checked again. If one of them references an object that is a primitive array or is classified as *array* or *entry*, the object gets classified as *head*. Otherwise, the object gets classified as *contained*.

²<http://ssw.jku.at/General/Staff/Weninger/AntTracks/ICPE18/CollectionHealthClassifier.java>

Instance Health. Instance health splits every heap object’s bytes into four different parts: (1) *primitive*, which encompasses primitive array elements and primitive fields (2) *header*, the memory consumed by the virtual machine (3) *pointer*, memory occupied by references between objects (4) *null*, memory reserved for pointers but set to null.

The classification for instance health has been reproduced as a user-defined transformer in AntTracks³. The source object gets transformed into four *virtual objects*, one per instance health part, and every part gets assigned its appropriate size (i.e., byte count). The amount of bytes of the *primitive* part can be calculated by iterating the type’s fields and filtering them for primitive types. The information about the *header* size (which depends on the VM architecture, as well as whether compressed oops are used) is stored in the symbols information generated alongside the trace file. Since an object’s pointer array contains one entry per pointer, either with the referenced object’s address or `-1` if the pointer is null, the bytes made up by *pointers* and *null* can also be easily calculated.

The judgment schemes presented by Mitchell and Sevitsky, i.e., the ways how to interpret combinations of both classifiers, can now also be analyzed in AntTracks by using both classifiers at the same time. Furthermore, they can be used in combination with any other classifier that AntTracks provides.

Mitchell and Sevitsky used “*the built-in facilities of Java virtual machines (JVM) to trigger writing a snapshot to disk*” [17]. Before being able to write an analysis tool for such heap snapshots, one must obtain knowledge about the binary file format, how to parse it, and how to combine the parsed data into a convenient data structure. Depending on the use-case, results also have to be presented graphically to the user to allow user-friendly manual analysis, which also may take up a significant amount of development time. Compared to that, the implementation of the two classifiers presented above took about two hours each, including writing unit tests (by checking the correct classification of known Java classes such as `HashMap`). The `classify` methods of both classifiers cover less than 150 lines of code (LOC). Therefore, we claim that writing *user-defined classifiers* takes less work, with regard to person hours as well as LOC. Additionally, AntTracks provides convenient visualization out-of-the-box and the possibility to combine the newly developed classifier with any other available classifier.

6 RELATED WORK

Current state-of-the-art tools share one common problem. Nearly all of them represent heap states (or the change of the heap over time) only as type histograms. No free selection of classification exists, not even to mention multi-level grouping. Even basic information such as an object’s allocation site is not available in many cases, since most tools rely on heap dumps that do not provide that level of detail. Still, some tools provide additional functionality such as pointer information on object level (plainly reconstructed from a heap dump).

The most basic approach supported by the Java Hotspot VM are the `-XX:+PrintClassHistogramBeforeFullGC` and `-XX:+PrintClassHistogramAfterFullGC` flags. They cause a class

³<http://ssw.jku.at/General/Staff/Weninger/AntTracks/ICPE18/InstanceHealthClassifier.java>

histogram to be printed to the console on every full GC. *JConsole* [18] can connect to a running Java application and retrieves data from its Java Management Beans. Due to the restricted functionality of the memory bean, it can only show the current heap memory consumption separated into eden space, survivor space, and old space. *jhat* [19] can be used to analyze a Java heap dump file which has previously been generated using the *jmap* tool. It starts a webserver that hosts the heap dump results and can be accessed via a webbrowser. Beside a type histogram, also the rootset (i.e., objects that are referenced by a GC root) can be shown. *Visual VM* [23] is a general performance monitoring tool for Java applications that provides memory analysis based on heap dumps. In addition to a type histogram, it allows users to analyze individual objects of a certain type, including functionality to follow an object's pointers and go to the referencing object. It is also able to calculate the retained set of objects. The retained set of an object X is the set of objects which would be removed when X is garbage collected. In addition to that, the *Eclipse Memory Analyzer (MAT)* [7] also allows users to analyze the application's dominator tree [15]. The *Netbeans profiler* [20] is just a slimmed down version of Visual VM and is integrated into the Netbeans IDE.

Other approaches such as the one presented by Aftandilian et al. [1] or De Pauw and Wim [22] focus on visualizing a heap state's object graph. To reduce the complexity of such graphs, certain reduction operations such merging, cutting, and so on, are applied. Such approaches may work well for pointer analysis, e.g., which types references which types, yet most of them lack the flexibility to take other properties into account, e.g., heap spaces or allocation sites.

A query technique that is integrated into some of the mentioned tools is the *Object Query Language (OQL)* [2, 5]. It has been developed by the Object Data Management Group and is an SQL-like query language used to query objects from object-oriented databases. The downside of OQL is its complexity, which results in the problem that no vendor implements the whole standard. For example, the Eclipse Memory Analyzer (MAT) as well as VisualVM only allow queries in the form of `SELECT <select clause> FROM <from clause> WHERE <where clause>`. *Where clauses* can be represented in our approach using filters, while *select clauses* can be represented using an object classifier. Multi-level grouping, as supported in our approach, is neither possible in MAT nor in VisualVM.

7 FUTURE WORK

The concept of object classifiers and multi-level grouping as well as their implementation in AntTracks opened a number of interesting ideas. This section will shortly introduce these ideas and point out possible ways how to approach them.

Extended Pointer Support in AntTracks. Currently, AntTracks provides only basic support for pointer analysis. For every object, it records the referencing and the referenced objects and makes them available for offline analysis. However, state-of-the-art tools [1, 16] often use advanced data structures such as dominator trees for analyzing whole pointer graphs. We plan to use similar data structures also in AntTracks to compute, for example, all objects that are reachable from a certain object (i.e., the transitive closure [24])

as well as the amount of memory that is kept alive by a specific object (i.e., the retained size).

Heap Diffing. Weninger et. al. [25] suggest heap diffing, i.e., analyzing how the heap changes over a certain time span, which is currently already supported to a certain level in AntTracks. The grouping and classification techniques that were described for heap states in this paper can partially also be applied to heap diffing. Extending classifiers with information about a source object's development over time, e.g., how a heap object's pointers changed over time, could further increase the potential application of heap diffing in combination with object classifiers.

Combined Tree Types. We showed that the memory consumption of a lossless classification tree is orders of magnitude higher than that of a lossy one. In a classification tree, often only a small subtree is of interest to the user. Since both classification tree types use node data structures inheriting from the same interface, they could be combined to only give lossless information for parts of the tree that are of higher interest to the user.

AntTracks DSL. To abstract from classifiers and their underlying programming language, the heap could also be analyzed by using a domain-specific query language. Such a language could, for example, be used to ask for the amount of objects of type T that were allocated at site S and survived at least n garbage collections. Based on our classifiers, we plan to develop such a language to provide even better support for expressing application-specific queries in a user-friendly way.

8 THREATS TO VALIDITY AND LIMITATIONS

Visualization of data in memory analysis tools is often strongly coupled with the kind of data that is collected and analyzed by those tools. Even though AntTracks collects more information about objects than most of the presented tools (e.g., only few tools collect allocation site information), the general classification principles using multi-level grouping and classification trees based on object classifiers and as well as AntTracks visualization features are not dependent on that amount of information. Only the number and the complexity of the classifiers that developers can implement is limited by the available information. The fewer source object properties are available, i.e., the less information the tool collects about heap objects, the less flexibility the developer has when it comes to writing classifiers. Assuming that AntTracks only collected type and heap space information for each object, we would still be able to provide the *Type classifier*, the *Object Kind classifier*, the *Space classifier* and so on as predefined classifiers, but due to the missing information, no *Allocation Site classifier* could be provided. Yet, all the available classifiers could still be freely combined, for example, by first classifying all objects by space and then by type, or first by object kind and then by space, or in any other possible combination. This outclasses the flexibility of the data aggregation and visualization techniques available in other tools presented in Section 6.

Similar to the limitation mentioned above, current pointer-based classifiers are restricted to adjacent objects via the from-pointer and to-pointer information. As explained in Section 7, new classifiers

may become possible as soon as AntTracks provides full object graph traversal and root pointer information.

To verify that the extra flexibility simplifies memory analysis, specifically that it facilitates detecting and resolving memory-related problems such as memory leaks, a user study is planned as future work. Technical metrics such as *task completion time* or *number of found memory leaks* and subjective metrics such as *user satisfaction* can be collected during the study, based on faulty benchmark implementations or industry applications.

A limitation of our current study is that we have not yet investigated, which combinations of classifiers are best for detecting specific memory-related problems. This is another topic to be tackled by the mentioned user study.

9 CONCLUSION

In this paper, we presented the domain-independent concepts of (user-defined) object classifiers and multi-level grouping, which are novel and general concepts for classifying large amounts of objects, processing them, and arranging their classification results as a tree for later analysis. Object classifiers are entities that classify objects based on a certain criterion derived from the objects' properties. Multi-level grouping is the process of applying multiple object classifiers to a collection of objects and grouping these objects based on the classification results. In contrast to single-level grouping, which results in a key-value map, multi-level grouping results in a classification tree. Such a tree can be visualized in various ways and allows a top-down, fine-grained manual data analysis by the user.

Various lossless and lossy *classification tree* data structures were presented and analyzed with respect to their performance, their memory consumption, and their ability to retain object identity. We showed that the lossy tree structures allow a tremendous reduction of memory overhead when accepting certain information loss in the classification tree.

We integrated the concept of object classifiers and multi-level grouping into the memory monitoring tool AntTracks, a tool that primarily focuses on helping developers to detect and understand memory anomalies, thus replacing its previous rigid classification scheme. Developers benefit from AntTracks's new ability to classify heap states based on any combination of classifiers, which distinguishes our approach from existing state-of-the-art tools. Furthermore, our tool supports *user-defined* object classifiers, i.e. it allows the user to write small, dynamically loaded source code snippets to classify heap objects based on arbitrary criteria. This may also be of interest to researchers who want to perform more general and experimental memory analyses. Our memory analysis approach opens new ways how AntTracks can be used and how memory can be analyzed, and its applicability has been shown in a quantitative and a functional evaluation.

ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria GmbH.

REFERENCES

- [1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program

- Understanding and Debugging. In *Proc. of the 5th Int'l. Symposium on Software Visualization (SOFTVIS '10)*. 53–62.
- [2] A. M. Alashqur, S. Y. W. Su, and H. Lam. 1989. OQL: A Query Language for Manipulating Object-oriented Databases. In *Proc. of the 15th Int'l. Conference on Very Large Data Bases (VLDB '89)*. 433–442.
- [3] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. 76–89.
- [4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dinkelage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*. 169–190.
- [5] R.G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez. 2000. *The Object Data Standard: ODMG 3.0*.
- [6] Technische Universität Darmstadt. 2012. DaCapoScala (last accessed October 10, 2017). <http://www.benchmarks.scalabench.org/modules/scala-benchmark-suite/>. (2012).
- [7] Andrew Johnson and Krum Tsvetkov. 2017. MAT - Eclipse Memory Analyzer (last accessed October 10, 2017). <http://www.eclipse.org/mat/>. (2017).
- [8] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. 31–38.
- [9] Maria Jump and Kathryn S. McKinley. 2009. Dynamic Shape Analysis via Degree Metrics. In *Proc. of the 2009 Int'l. Symposium on Memory Management (ISMM '09)*. 119–128.
- [10] Philipp Lengauer, Verena Bitto, Florian Angerer, Paul Grünbacher, and Hanspeter Mössenböck. 2013. Where Has All My Memory Gone?: Determining Memory Characteristics of Product Variants Using Virtual-machine-level Monitoring. In *Proc. of the Eighth Int'l. Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '14)*. Article 13, 13:1–13:8 pages.
- [11] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*. Article 4.
- [12] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conference on Performance Engineering (ICPE '15)*. 51–62.
- [13] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '16)*. 249–260.
- [14] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proc. of the 8th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '17)*. 3–14.
- [15] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141.
- [16] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks Using Graph Mining on Heap Dumps. In *Proc. of the 16th ACM SIGKDD Int'l. Conference on Knowledge Discovery and Data Mining (KDD '10)*.
- [17] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proc. of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. 245–260.
- [18] Oracle. 2017. JConsole (last accessed October 10, 2017). <https://docs.oracle.com/javase/9/troubleshoot/diagnostic-tools.htm#JSTGD174>. (2017).
- [19] Oracle. 2017. jhat (last accessed October 10, 2017). <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jhat.html>. (2017).
- [20] Oracle. 2017. Netbeans profiler (last accessed October 10, 2017). <https://profiler.netbeans.org/>. (2017).
- [21] Oracle. 2017. OpenJDK HotSpot group (last accessed October 22, 2017). <http://openjdk.java.net/groups/hotspot/>. (2017).
- [22] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proceedings of the 13th European Conf. on Object-Oriented Programming (ECOOP '99)*. 116–134.
- [23] Jiri Sedlacek and Tomas Hurka. 2017. Visual VM - All-in-One Java Troubleshooting Tool (last accessed October 10, 2017). <https://visualvm.github.io/>. (2017).
- [24] R. Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. 114–121.
- [25] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '17)*. 357–360.

3.2 GC Roots and Closures

This section includes the paper [317] that describes (1) how we modified the AntTracks VM to also collect information about garbage collection roots and (2) how we use object reference information to calculate closures around heap objects and heap object groups to detect suspicious object ownership.

Paper:

Markus Weninger, Elias Gander, Hanspeter Mössenböck:

Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, ManLang 2018, Linz, Austria, September 12-14, 2018.

Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring

Markus Weninger
Institute for System Software,
CD Labor MEVSS,
Johannes Kepler University
Linz, Austria
markus.weninger@jku.at

Elias Gander
CD Labor MEVSS,
Johannes Kepler University
Linz, Austria
elias.gander@jku.at

Hanspeter Mössenböck
Institute for System Software,
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Complex software systems often suffer from performance problems caused by memory anomalies such as memory leaks. While the proliferation of objects is rather easy to detect using state-of-the-art memory monitoring tools, extracting a leak's root cause, i.e., identifying the objects that keep the accumulating objects alive, is still poorly supported. Most state-of-the-art tools rely on the dominator tree of the object graph and thus only support single-object ownership analysis. Multi-object ownership analysis, e.g., when the leaking objects are contained in multiple collections, is not possible by merely relying on the dominator tree. We present an efficient approach to continuously collect GC root information (e.g., static fields or thread-local variables) in a trace-based memory monitoring tool, as well as algorithms that use this information to calculate the *transitive closure* (i.e., all reachable objects) and the *GC closure* (i.e., objects that are kept alive) for arbitrary heap object groups. These closures allow to derive various metrics for heap object groups that can be used to guide the user during memory leak analysis. We implemented our approach in AntTracks, an offline memory monitoring tool, and demonstrate its usefulness by comparing it with other widely used tools for memory leak detection such as the Eclipse Memory Analyzer. Our evaluation shows that collecting GC root information tracing introduces about 1% overhead, in terms of run time as well as trace file size.

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**; *Software defect analysis*; • **Information systems** → *Clustering and classification*; • **Theory of computation** → *Data structures design and analysis*;

KEYWORDS

Memory Monitoring, Memory Leak, Pointer Analysis, Garbage Collection, Graph Closure,

ACM Reference Format:

Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *15th International Conference on Managed Languages & Runtimes (ManLang'18)*, September 12–14, 2018, Linz, Austria. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3237009.3237023>

1 INTRODUCTION

Modern programming languages such as Java or C# rely on garbage collection to relieve the programmer from freeing allocated memory manually. The garbage collector (GC) tries to free heap space by removing dead objects that are not reachable from root objects anymore. Examples for root objects are objects referenced by GC roots such as static fields or thread-local variables. Since these objects cannot be reclaimed by the GC, they also keep alive all other objects that are directly or indirectly referenced by them. This reveals one of the drawbacks of garbage collection. Since programmers are no longer required to free memory manually, they tend to use object allocations more carelessly. However, even in garbage-collected languages, careless handling of memory can lead to anomalies such as memory leaks. A memory leak occurs when objects that are not needed anymore are still unintentionally reachable and can therefore not be garbage collected.

Detecting the presence of a memory leak is often relatively easy. A simple chart displaying an application's growing memory usage may be enough to detect such a leak. Yet, it is difficult to track down a memory leak's root cause, i.e., to identify which objects are leaking and which objects are responsible for that by keeping the leaking objects alive. Most state-of-the-art memory monitoring tools analyze the heap based on the object graph in conjunction with its dominator tree [25]. The object graph is a representation of the heap state, where each object is represented as a node, and the references between objects are represented as edges. The dominator tree of an object graph describes a *keeps-alive relationship* between the objects. If an object *A* dominates an object *B* and *A* is collectible by the GC, also *B* is collectible. While the dominator tree may help to find memory leaks where a single object is responsible, e.g., when all leaking objects are contained in a single large list, it fails to provide insight in situations where multiple objects are keeping objects alive (e.g., in multiple collections) [6].

AntTracks [4, 21–23] is a memory monitoring tool for Java based on the Java Hotspot™ VM [30]. During an application's execution, it records various events such as object allocations, object moves during garbage collection, or pointer information. Based on such

ManLang'18, September 12–14, 2018, Linz, Austria

© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *15th International Conference on Managed Languages & Runtimes (ManLang'18)*, September 12–14, 2018, Linz, Austria, <https://doi.org/10.1145/3237009.3237023>.

trace files, the heap can be reconstructed for any garbage collection point in time. For inspecting the live objects at such points, Weninger et al. [42, 43] presented the concept of object classifiers and multi-level grouping, which enable the user to classify and group heap objects on multiple levels based on arbitrary grouping criteria.

In this paper, we extend their work by presenting a novel approach to collect and use information about GC roots and the object graph in order to guide users in finding the root causes of memory leaks. First, our approach encompasses an efficient technique to collect GC root information of different kinds (e.g., of static fields or thread-local variables) in a trace. Then, we show how to use this information to calculate object closures. AntTracks is able to calculate closures for arbitrary heap object groups, not just for single objects as other approaches that rely on dominator-tree-based analysis. The *transitive closure* encompasses all objects reachable from the given object group. The *GC closure* encompasses all objects kept alive by the given object group, i.e., those objects that could get garbage collected if the given object group was freed. Based on these closures, metrics such as the *transitive size* and the *retained size* can be calculated. We show how these metrics, in combination with AntTracks's user-driven classification system, can be used to detect memory leaks. Finally, we show in a quantitative evaluation based on three different well-known benchmark suites that collecting GC root information introduces about 1% overhead in terms of run time and trace file.

Thus, our scientific contributions are

- (1) a concept to integrate information about GC roots into a trace-based memory monitoring tool such as AntTracks,
- (2) algorithms to calculate the transitive closure and the GC closure of a single heap object as well as of heap object groups to derive meaningful metrics,
- (3) various techniques to use this information in top-down and in bottom-up memory analysis,
- (4) a quantitative evaluation of the tracing overhead and a functional evaluation of our approach based on typical memory anomaly detection use cases.

The paper is organized as follows: Section 2 provides the background of our work, Section 3 describes our approach as well as its concepts and techniques, Section 4 provides details on the implementation of these concepts in AntTracks, Section 5 presents a quantitative and a functional evaluation, Section 6 discusses related work, Section 7 outlines possible future work and discusses threats to validity, and Section 8 concludes the paper.

2 BACKGROUND

AntTracks consists of the *AntTracks VM*, a virtual machine based on the Java Hotspot™ VM [30], and the *AntTracks Analyzer*, a memory analysis tool. The AntTracks VM records memory events into trace files, which can then be analyzed offline with the tool. Since the concepts presented in this paper have been integrated into AntTracks, it is essential to understand AntTracks's architecture and workflow alongside basic garbage collection mechanisms.

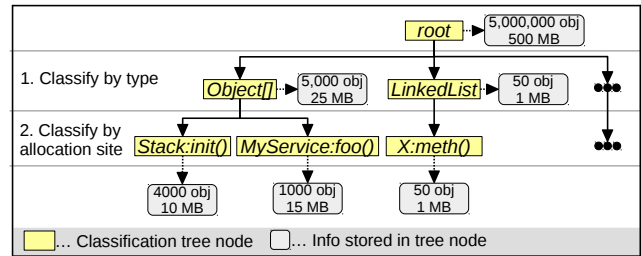


Figure 1: A heap state, consisting of 5 million heap objects, first classified by type followed by allocation site.

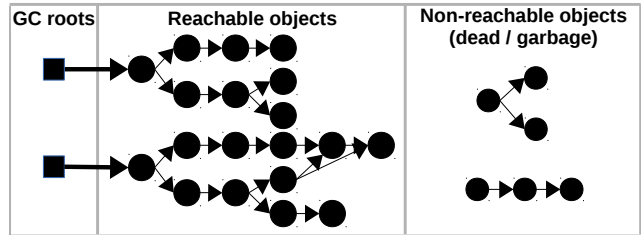


Figure 2: All objects directly or indirectly reachable by GC roots are considered live. Other objects are considered dead or garbage, and may be collected by the GC.

2.1 AntTracks VM: Trace Recording and Reconstruction

The AntTracks VM records memory events, e.g., events for object allocations and object movements executed by the GC, and writes them into trace files. After parsing such a trace file, the AntTracks Analyzer provides an overview of the memory behavior over time and can reconstruct the heap state for every garbage collection point by incrementally processing the events in the trace. A heap state is the set of heap objects that were live in the monitored application at a certain point in time. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by.

2.2 AntTracks Analyzer: Memory State Analysis

The AntTracks Analyzer uses user-defined object classifiers and multi-level grouping [42, 43] to enable user-driven heap state analysis. Object classifiers classify heap objects based on certain criteria such as their type, their allocation site, their allocating thread, and so on. For example, the *Type* classifier classifies a heap object based on its type's name, e.g. `java.util.HashMap`. Multi-level grouping is the process of applying multiple classifiers to a collection of objects (i.e., to a heap state), and grouping these objects based on the classification results into a hierarchical classification tree.

A typical example in AntTracks is to first group all heap objects by their types (using the *Type* classifier) and then by their allocation site (using the *Allocation Site* classifier). Figure 1 shows such a classification tree. Yellow rectangles represent tree nodes, and gray rounded rectangles represent data about all heap objects that were classified by the respective tree branch (basically the number of objects and the number of bytes).

2.3 Garbage Collection Roots

Certain objects in the heap are so-called *root objects*. Root objects must not be collected during a garbage collection, and neither must objects be collected that are directly or indirectly reachable from root objects. Whether a heap object is a root object or not is determined by the fact whether at least one *GC root* is referencing the object. There are different kinds of GC roots in Java (and in most other object-oriented languages), the most important ones are:

- *Local variables of threads*: Local variables reside on the call stack of a thread and may reference objects on the heap. Therefore, threads and all objects referenced by local variables of threads are always considered root objects.
- *Static variables*: Static fields of loaded classes are GC roots, i.e., objects referenced by static fields are considered root objects.

Figure 2 shows an example of how GC roots keep reachable objects alive whereas non-reachable objects are eligible for garbage collection.

2.4 Running example

For the algorithms in the following sections we will use a singly-linked list as a running example (see Listing 1). A singly-linked list is represented by its root object of type `LinkedList`. If a list contains at least one object, its `LinkedList` instance points to the head node of the list, an instance of `Node`. Every node may point to another node (its successor) and must point to an instance of `Data`. A `Data` instance contains some integer value and may point to another `Data` instance.

A heap state can be represented as an object graph, i.e., a directed graph in which the nodes correspond to heap objects and references between objects are the edges. Figure 3 shows the visualization of an object graph of two `LinkedList` instances and their referenced objects.

```

class LinkedList {
    private Node head;
    // ...
}
class Node {
    Node next;
    Data data;
    Node(Data d) { data = d; }
    // ...
}
class Data {
    int value;
    Data other;
    Data(int v) { this(v, null); }
    // ...
}
    
```

Listing 1: Code example for a singly-linked list

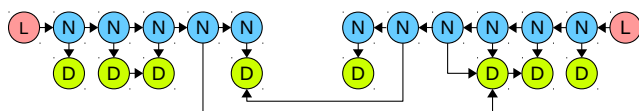


Figure 3: Object graph of two `LinkedList`s (L, red), one with five `Nodes` and one with six `Nodes` (N, blue) that point to eight `Data` instances (D, green).

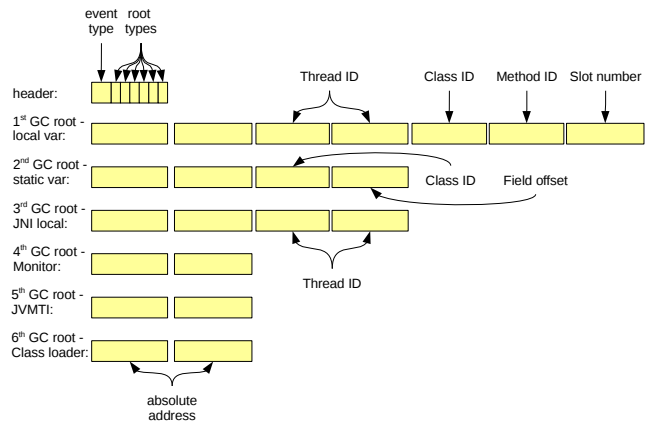


Figure 4: Example of a GC root event (with a size of $22 * 4 = 88$ bytes) that contains information about a local variable, a static field, a JNI GC root and 3 other GC roots without additional information.

3 APPROACH

In this section, we show which information can be retrieved from GC roots and how it can be integrated into our trace-based memory monitoring approach. We further present algorithms to calculate the transitive closure and the GC closure for arbitrary heap object groups. Finally, we discuss how closures and GC root information can be used as a guidance to users when tracking down memory anomalies and memory leaks.

3.1 Retrieving GC Root Information

There are two kinds of information that can be retrieved from GC roots: (1) information about the GC roots themselves and (2) information about the references between the objects that are directly or indirectly reachable from the GC roots. Approaches that rely on snapshot-based analysis create heap dumps, which contain information about the objects that were live at a certain point in time as well as the references between them and GC root information. However, if we are not just interested in the heap state at a single point in time, but in the development of the heap over time, heap dumps may prove insufficient.

Instead of creating multiple heap dumps, which would introduce enormous run time overhead, continuous tracing approaches such as AntTracks produce trace files. These allow the offline reconstruction of heap states for arbitrary garbage collection points in time. Trace files contain a sequence of events that have a certain encoding and are only allowed in a certain order. Lengauer et al. [21, 22] present a general event format to trace objects and the references between them. In the following, we propose an additional *GC root event* to trace information about GC roots and root objects.

GC root events are written before any other event at the start of every garbage collection. The format of a GC root event is shown in Figure 4, where each block represents a word of 4 bytes. Like every other event in AntTracks's event format, a GC root event starts with a 1-byte event type. The remaining 3 bytes of the first word are filled with six 4-bit numbers, each of which indicates the type of one GC root that is encoded in the event (see Table 1 for the different GC root types) or is set to 0 if less than six GC roots

ID	Type	Description	Info encoded in event
1	Class Loader	GC root referencing a class loader	-
2	Static Field	GC root representing a static field referencing a heap object	class id, field offset
3	Thread	GC root referencing a thread	thread id
4	Local variable	GC root representing a local variable referencing a heap object	thread id, class id, method id, slot number
5	Code Blob	GC root referencing a code object (e.g. a JIT-compiled method)	class id, method id
6	JNI	GC root referencing a heap object created via JNI	thread id
7	JVMTI	GC root referencing a heap object created via JVMTI	-
8	Monitor	GC root referencing a monitor object used for synchronizing	-
9	Management	GC root referencing internal objects used by MXBeans	-
10	Others	GC root referencing other JVM internal object (e.g., profilers)	-

Table 1: The different GC root types in the JVM.

are encoded in the event. Thus, the number of GC roots that can be encoded in a single GC root event is limited to six. For every GC root encoded in the event, the address of the referenced root object as well as additional information (depending on the type of the GC root) is appended after the header word (see Figure 4).

The types of the different GC roots encoded in the header word are necessary to correctly interpret the given GC root's information. As shown in Table 1, depending on the type of each GC root, additional information is added to the event's payload (the size of the different IDs is determined by the JVM):

- A *static field root* adds a 4-byte class ID and a 4-byte field offset.
- A *thread root* adds an 8-byte thread ID.
- A *local variable root* adds an 8-byte thread ID (i.e., the thread that holds the local variable), a 4-byte class ID (i.e., the class in which the local variable is defined), a 4-byte method ID (i.e., the method in which the local variable is defined) and a 4-byte slot number (i.e., the local variable stack index).
- A *code blob root* adds a 4-byte class ID and a 4-byte method ID.
- A *JNI local root* adds an 8-byte thread ID.

The connection between a thread's ID and its name is established by a separate event that is recorded whenever a new thread is started. This information can later be used to resolve a thread ID into the thread's name. Similarly, class IDs, method IDs, field offsets and slot numbers can be resolved to their identifiers and type signatures using symbol information that is written to a separate *symbols file* during program execution.

Based on local variable GC roots, the call stacks of all threads at the time of the garbage collection can be reconstructed. This is possible because (1) each local variable root stores the method and thread they belong to and (2) they are written in the same order as they are encountered when traversing their thread's stack frames from top to bottom. This excludes stack frames that do not contain at least one local variable.

The amount of information differs among GC root types, thus the total event size is variable. However, the maximum size of a GC root event is 43 words (336 bytes) which is reached in the case of 6 local variable GC roots.

Since GC roots are recorded at the start of every garbage collection, their referenced objects may be moved during garbage collection. AntTracks also records these object movements. When encountering a matching move event while parsing a trace file, GC roots have to be updated to the move's destination address.

3.2 Heap Object Closures

In general, a heap object closure is the set of heap objects that are directly or indirectly reachable from a given heap object or from a group of heap objects. A heap object closure may contain *all* reachable objects (i.e., the transitive closure) or only those that satisfy certain criteria (e.g., in the case of the GC closure). In this section, we show how to calculate closures for arbitrary object groups. This allows us to detect leaking objects that are kept alive, even by more than one object. Metrics derived from the closures such as the *retained size* can be displayed to help and guide users during heap state analysis. Closure calculations, as well as metrics derived from these closures, have been integrated into AntTracks. Nevertheless, these techniques could also be integrated into other heap profiling tools such as Elephant Tracks [39], as long as they are able to reconstruct heap object graphs, i.e., they record the heap objects themselves, pointers between them, as well as garbage collection roots.

3.2.1 Transitive Closure. The transitive closure [20, 41] of a single node in a graph is made up of the node itself and all other nodes that are directly or indirectly reachable from this node following all edges. Its calculation has been well studied, e.g., by Eve and Kurki-Suonio [13].

Instead of calculating the transitive closure of a single object, we argue that it is also useful to calculate the transitive closure of an object group. We call the objects for which we calculate a closure the *closure root objects* (which are not to be confused with *root objects*, i.e., objects that are referenced by GC roots). The transitive closure of an object group is made up of the closure root objects themselves and all other objects that are directly or indirectly reachable from at least one of the closure root objects. Figure 5 shows the transitive closure (gray background), first for a single closure root object (thick border), and second for a group of two closure root objects. The transitive closure is calculated based on the pseudocode presented in Listing 2.

Section 4.2 discusses how the transitive closure algorithm is computed in AntTracks based on AntTracks's internal heap state data structure.

The transitive closure can be used to detect objects that reference an unexpectedly large amount of other objects since it describes how many objects are *reachable* from the given closure root objects. Analogously, an object group with a small transitive closure, i.e., an object group that does not reach many other objects, can never be the root cause of a memory leak. In AntTracks, heap objects are separated into object groups that get arranged in a classification

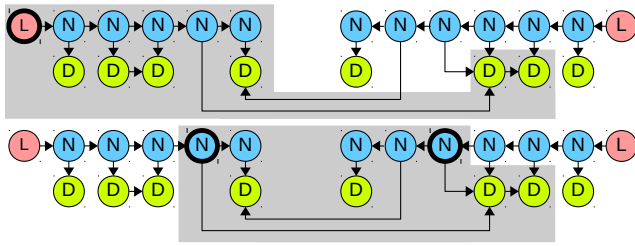


Figure 5: Two examples showing the transitive closures when using the thick bordered object(s) as closure root objects.

tree based on their classification results. If the user is only interested in finding memory leak roots, i.e., objects that keep a lot of other objects alive, tree nodes of object groups with small transitive closures can be hidden and excluded from further analysis. This reduces the tree's complexity, speeds up further computations, and thus eases the overall analysis process. Nevertheless, a large transitive closure does not automatically identify a leaking data structure or object group. Just because objects are reachable from a given set of closure root objects does not mean that only the closure root objects keep them alive. Therefore, the transitive closure can be used to further calculate the *GC closure*, a closure that describe the closure root objects' *ownership* over other objects.

3.2.2 *GC Closure / Retained Closure.* The transitive closure contains all objects reachable from the closure root objects. In contrast, the GC closure contains all objects that (1) are reachable from the given closure root objects *and* (2) could be garbage collected if all closure root objects were collected. Thus, the GC closure describes *object ownership*, i.e., all objects that are kept alive by a given set of closure root objects. Being able to detect heap object groups that keep large amounts of other objects alive bears large potential in helping users to resolve memory leaks.

To calculate the GC closure of a *single object*, the object graph's dominator tree can be used. In graph theory, a node *d* dominates a node *n* if every path from the root to *n* must pass through *d* [40]. Figure 6 shows the dominator tree of a sample object graph of two singly-linked lists. Some of the data is shared between the two lists, i.e., *D3*, *D4* and *D5*, which results in those three objects not having a dominator. For example, assuming that node *N8* could be collected by the garbage collector (by removing all references to it), all child objects in its dominator subtree (i.e., *D8*, *N9*, and *N10*) could be

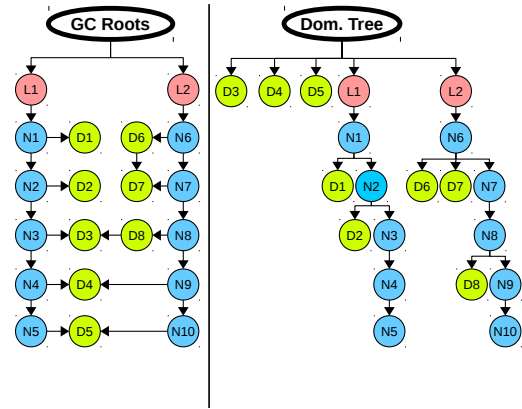


Figure 6: Object graph of two singly-linked lists and its dominator tree.

collected, too. However, it is not possible to use the dominator tree to answer which objects could be freed if a certain object group, e.g., *N4* and *N9*, would be freed (which would be *N5* and *N10*, but also *D4* and *D5*).

While the dominator tree and its algorithms are well-studied, they are only suited to analyze single nodes and to detect the maximum *unique ownership* [27] within a graph, e.g., what happens if one specific heap object could be freed by the GC. We are therefore presenting a new algorithm that is able to calculate the GC closure for arbitrary closure root object groups.

Simple Approach. Our approach assumes that an heap object group's transitive closure is already known, e.g., by using the algorithm presented in Section 3.2.1. This transitive closure can then be reduced to the GC closure by following these steps:

- (1) The initial GC closure is set to the transitive closure.
- (2) To determine which objects could be freed if the closure root objects get freed, we have to simulate that the closure root objects are not reachable from root objects anymore. Thus, we ignore all references to these objects (i.e., assuming that all references to the closure root objects have been set to null).
- (3) Then, the heap is recursively traversed, starting at the heap's root objects (i.e., the objects directly referenced by GC roots), visiting every reachable object exactly once. If the currently visited object is part of the GC closure (which has been initialized to the transitive closure, see Step 1), the current object and all objects reachable from it are removed from the closure. This is done because these objects would still be reachable from the heap's root objects and thus be kept alive, even if the closure root objects would be freed. Visited and removed objects are marked to avoid processing them multiple times.

At the end of this algorithm, the GC closure contains only those objects that are not referenced by GC roots from outside the transitive closure. The object in the GC closure are the objects that could be freed if the closure root objects were released.

A major problem with this approach is that its complexity depends on the object graph size, i.e., the number of objects in the heap, since the heap traversal starts at the root objects. Yet, with

```

// called with closure root objects as initial work list
transitiveClosure(List workList) {
    List closure = new List();
    mark all objects in workList as visited;
    foreach (obj in workList) {
        add obj to closure;
        foreach (child referenced by obj) {
            if(child is not yet marked as visited) {
                mark child as visited;
                add child to workList;
            }
        }
    }
    return closure
}
    
```

Listing 2: Pseudocode calculating the transitive closure for a given closure root object group

minor adjustments to the algorithm, the complexity can be reduced to only depend on the object group's transitive closure size.

Improved Approach. The performance problems of the simple approach can be tackled by using the following technique. When a heap state is reconstructed from a trace file, we traverse the object graph once (instead of traversing the object graph on every GC closure calculation in the simple approach). Starting at the GC root objects, every visited object is marked. This allows us to identify all objects either as live (if they have been visited) or as dead (if they have not been visited). Since this additional information can be stored as a single bit per object (i.e., 0 if the object is not reachable from any GC root or 1 if it is) the additional memory overhead is negligible. For example, this information would take up around 12.5 MB for a heap state of 100,000,000 objects). The improved approach works as follows:

- (1) The initial GC closure is set to the transitive closure.
- (2) To determine which objects could be freed if the closure root objects get freed, we have to simulate that the closure root objects are not reachable from root objects anymore. Thus, we ignore all references to these objects (i.e., assuming that all references to the closure root objects have been set to null).
- (3) The GC closure (which has been initialized to the transitive closure) is recursively traversed, starting at the closure root objects, visiting every object in the closure exactly once. The current object is checked for the following criteria:
 - Is it directly referenced by a GC root?
 - Is it easily be checked since we know every GC root.
 - Is it referenced by a live object that is not part of the closure?

AntTracks enables access all objects that reference a given object (pointed-from analysis). For each of these referencing objects it can be checked whether it is part of the closure, and whether it has been marked as live during the heap state reconstruction.

If at least one of the mentioned criteria holds, the current object and all objects reachable from it are removed from the closure. Visited and removed objects are marked to avoid processing them multiple times.

Instead of having to traverse the whole heap, the algorithm's complexity now only depends on the transitive closure's size. Figure 7 shows how objects that are reachable from live objects outside the transitive closure have to be removed to reduce the transitive closure to the GC closure.

3.3 Metrics

A single object's shallow size is the size of the object itself. This encompasses the object's header and its data, without taking into account any referenced objects. The shallow size of an object group is the sum of the shallow sizes of all contained objects. Using the transitive closure, the *transitive object count* (also called *deep object count*) as well as the *transitive size* (also called *deep size*) can be calculated. The *transitive object count* is the number of objects contained in the transitive closure, while the *transitive size* is the sum of the shallow sizes of all objects in the transitive closure. Similarly, using the GC closure, the *retained object count* as well as

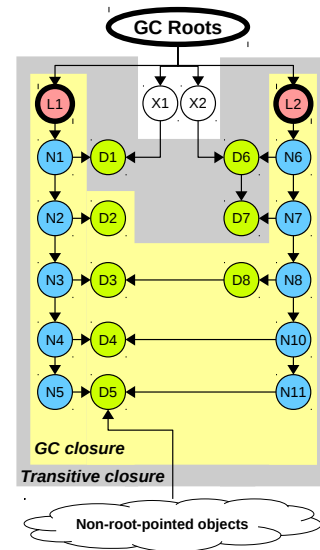


Figure 7: GC closure for L1 and L2, derived from the transitive closure and GC root information.

the *retained size* can be calculated. The *retained object count* is the number of objects contained in the GC closure, i.e., the number of objects that could be freed if the closure root objects were released, while the *retained size* is the sum of the shallow sizes of all objects in the GC closure.

As explained in Section 2, AntTracks applies user-specified object classifiers to aggregate heap objects in a classification tree (see Figure 1). Without information about an object group's transitive closure and GC closure, only the group's *object count* and *shallow size* could be calculated and displayed to the user. While this information is good enough to learn about the system under investigation (e.g., about the most frequently allocated types, the hot allocation sites, the object allocations per thread, etc.), it is not well suited to track down memory leaks. So far, the users' decisions on which object groups to analyze in more detail (e.g., by applying further classifiers) often strongly depended on prior knowledge about the system under analysis, e.g., by having an educated guess on what to look for. For example, a user unaware of the internals of a given system may not suspect a possible memory leak when discovering that there was a single instance of type A with a small shallow size. However, knowing that objects of type A are very complex (i.e., may keep a lot of other objects alive) and are not expected to be alive in the given heap state would lead to further investigation.

Thus, we extended AntTracks to also display the deep size as well as the retained size for every tree node in a classification tree. Even without knowing anything about the system under investigation, the user is now able to detect object groups that keep a lot of other objects alive. An evaluation of this feature will be described in Section 5.1.

3.4 Utilizing GC Root Information

In some cases, one might be aware of a group of objects that occupy a large portion of the heap. For example, by analyzing multiple heap states, we may detect that objects of a certain type that are

allocated at a certain allocation site accumulate over time. To free their memory, these objects must be made eligible for garbage collection. This can be done either by freeing all root objects that reference the object group or by cutting all connections to them. Therefore, bottom-up analysis is used to detect possible ways to free memory occupied by a given object group by inspecting the relation between the object group and its referencing root objects. For example, we may detect a suspiciously large object group. By following the object references from the object group back to the GC roots we may detect that all these objects are stored in a list, which in turn is referenced directly or indirectly by multiple GC roots. Ideally, we are able to remove the list head, and in turn the whole object group becomes unreachable and can thus be garbage collected. If this is not possible, suitable cutting points to remove single elements from the list have to be found.

3.4.1 Root Object Classification. Assume that we detect a suspicious object group. As a first step, we obtain all root objects that reference the object group. Since not every object of the suspicious object group must be referenced by every found root object, we have to detect those root objects that own most of them. This can be done by further analyzing the found root objects, e.g., by applying multiple classifiers on them (e.g., classifying them by their types, packages, allocation sites, etc.). Since the classified object groups can be sorted by their size or by their retained size, we can try to find object groups that are small but still keep a large number of other objects alive. This small number of root objects is then feasible to be traced and dealt within the code.

3.4.2 Bottom-up Visualization. Alternatively, instead of analyzing the root objects directly, one can also trace and visualize all paths from the suspicious object group to their root objects. These paths may reveal possible cutting points to make (parts of) the object group unreachable and thus eligible for garbage collection. To facilitate the task of locating high-impact cutting points, the paths can be displayed as a graph with objects as nodes and their pointers as edges. Additionally, since object groups can be very large, node aggregation would be necessary in many cases. The goal of node aggregation is to combine multiple objects into a single node by detecting typical reference patterns between them. This could, for example, be achieved by merging nodes that belong to the same data structure, which is a goal of our future work. Node aggregation decreases the complexity of the graph and thus enables users to draw meaningful conclusion from it more easily.

4 IMPLEMENTATION

4.1 Retrieving GC Root Information

AntTracks uses a modified Java Hotspot™ VM to detect events (e.g., object allocations) and to write information about these events into trace files. GC root information gets written at the beginning of every garbage collection. Since there are different types of GC roots, various VM-internal data structures are used to retrieve this information. The `ClassLoaderDataGraph` contains information about classes that are loaded by an application's class loaders. Other data structures include the `SystemDictionary` and the `Universe`. These data structures are needed to retrieve information about static fields, since static fields are stored inside Java class instances.

To collect information about thread-local variables, we iterate all Java threads and extract each thread's stack trace. Each frame in the stack trace contains a `StackValueCollection` that allows to access information about the frame's local variables. Also, the thread object itself is marked as a root object. Furthermore, we handle `CodeBlobs` as GC roots. These may be JIT-compiled static references to certain addresses. `JNIHandles` and `JvmtiExport` are used to detect objects that are kept alive via JNI and JVMTI. Other types of GC roots are `ObjectSynchronizer` (i.e., monitors), `FlatProfiler` (i.e., a JVM profiling tool), and `Management` (i.e., objects used by MXBeans).

4.2 Closure Algorithms

Section 3.2 presented algorithms to calculate the transitive closure and the GC closure for arbitrary object groups. Implementing these algorithms efficiently is crucial for their application in end-user applications such as AntTracks Analyzer. After evaluating various approaches, we decided to use `BitSets` for the representation of closures. A `BitSet` is a vector of bits that is indexed by a nonnegative object number and keeps track of which objects are part of the closure.

To be able to use `BitSets`, we had to adjust AntTracks's heap data structure. It is now organized in such a way that objects can be identified by a unique number that is used as their index in the bit sets. The heap object at index 0 is the object with the lowest address, while the object at index n (where n is the number of objects in the heap) is the object with the highest address.

Another advantage of bit sets is their ability to be combined using logical operators such as `and` or `or`. For example, using these operators, we do not need to calculate the transitive closure for every node in the classification tree, but only for leaf nodes. Imagine classifying heap objects first by type and then by allocation site. This may result in a classification tree node for two objects of type A (intermediate node), one allocated at allocation site x (leaf node) and one at allocation site y (leaf node). It is sufficient to calculate the transitive closures for the x node and the y node based on the object graph, since the transitive closure for node A is just the union of the two closure for x and y . The union of these closures can be obtained by `or`-ing their bit sets.

4.3 Classifiers

In addition to the metrics presented in Section 3.3, we extended AntTracks by a number of new classifiers, which can be freely combined with any other existing classifier to analyze a heap state.

4.3.1 Directly-GC-Rooted Classifier. This classifier categorizes heap objects based on the GC roots by which they are directly referenced and splits them into multiple groups (e.g., *Root: Static field* or *Root: Thread-local variable*). Each of these groups is then split into further subgroups, depending on the root type, to provide further information. For example, objects in the group *Root: Static field* are further split by the static fields' classes, and then by the static fields themselves (as shown in Figure 8). The classifier can also be configured to only show variables (static fields, thread-local variables and JNI locals) instead of all roots. In addition, the user can chose if objects that are only indirectly referenced by GC roots or not reachable from GC roots at all should be shown in a separate group or be completely hidden.

Name	Objects		Retained Size [bytes]	
	#	%	#	%
Overall	89,739	100.0	2,771,168	100.0
Not directly referenced by any variable	89,406	99.6	2,667,008	96.2
Root: Static field	318	0.4	2,629,224	94.9
Class: jku.antracks.example.gcrootclassifier.Service	2	0.0	2,450,600	88.4
Field: java.util.List names	1	0.0	1,363,560	49.2
Field: java.util.Map map	1	0.0	1,087,040	39.2

Figure 8: A heap state in AntTracks, classified using the directly rooted classifier.

Name	Objects		Retained Size [bytes]	
	#	%	#	%
Overall	89,739	100.0	2,771,168	100.0
java.lang.Integer	30,000	33.4	480,000	17.3
char[]	21,143	23.6	909,776	32.8
java.lang.String	21,129	23.5	1,382,672	49.9
Root: Static field	20,849	23.2	1,362,032	49.2
Class: jku.antracks.example.gcrootclassifier.Service	20,000	22.3	1,279,200	46.2
Field: java.util.List names	20,000	22.3	1,279,200	46.2

Figure 9: A heap state in AntTracks, classified using the type classifier followed by the indirectly rooted classifier.

Figure 8 shows an example for the use of the *directly-GC-rooted* classifier in AntTracks. The first line (with the key *Overall*) shows how many objects the heap contains. Its first child (with the key *Not directly referenced by any variable*) contains the 89.406 objects that are not directly root-pointed by a variable. The second child (with key *Root: Static field*) contains 318 objects that are directly root pointed by a static field. Two of them are referenced from a static field in the class `jku.antracks.example.gcrootclassifier.Service` (4th line), one by the field names (5th line) and one by the field map (6th line).

4.3.2 Indirectly-GC-Rooted Classifier. Similar to the *directly-GC-rooted* classifier, the *indirectly-GC-rooted* classifier categorizes heap objects based on the GC roots by which they are referenced. However, instead of only taking direct references into account, also indirectly referencing GC roots are considered. The classifier can also be configured to only show variables instead of all roots, as well as to whether objects that are not root-pointed at all should be shown in a separate group or be completely hidden.

This classifier is especially useful when the user encounters an unexpectedly large group of objects that was not assumed to be alive or when such a group has been growing over time. In such situations the *indirectly-GC-rooted* classifier can be used to find out which root objects are keeping them alive. To improve performance, it is also possible to cut the calculation as soon as the first referencing root object is encountered.

Figure 9 shows an example for the use of the *indirectly-GC-rooted* classifier, following the type classifier. 21,129 String objects exist at the given point in time, of which 20,849 are reachable from static fields. 20,000 of these strings are reachable from the static field names in the class `jku.antracks.example.gcrootclassifier.Service`.

4.4 Object Group Inspection Window

The classification mechanism in AntTracks produces object groups that share certain properties based on the selected classifiers (e.g., objects of the same type allocated at the same allocation site). Beside

further classifying such object groups (e.g., extending the classification tree by further splitting the objects by their allocating thread), a given object group can also be inspected in more detail in an *object group inspection window*.

A classification tree visualizes a particular heap state by showing the object count, the shallow size, the deep size, and the retained size of each object group. This is well suited for a fast overview. It either enables users to detect small groups of objects with large retained sizes (most probably heads of larger data structures) or large groups of objects (most probably contained in other data structures). The aim of the inspection window is now to provide more detailed information about a particular object group.

First of all, the view shows all root objects from which a selected object group can be reached and which keep the object group alive. These root objects can then be further classified (*root object classification*) or the paths to them can be visualized (*bottom-up visualization*) as described in Section 3.4. Second, the inspection window includes another classification tree showing the objects in the group’s transitive closure or in its GC closure. Again, classifiers can be applied to assign the objects in the closures to the branches of the tree (e.g., according to the object types in the closures). This feature might prove useful in helping user to understand the ownership relations between the inspected object group and the closure objects.

5 EVALUATION

To evaluate the usefulness of our analyses we show how one can use the AntTracks Analyzer to detect memory leaks and compare it to a dominator-tree-based state-of-the-art memory monitoring tool, namely the Eclipse Memory Analyzer (MAT) [15].

We also evaluate the overhead that is caused by recording the GC root data in terms of *run time* and *trace file size*. All analyses were performed on well-known benchmarks from the DaCapo suite¹ [5] (version 9.12-bach), DaCapo Scala suite² (version 9.12-bach) and the SPECjvm2008 suite³ (version 1.01).

Setup. All measurements were run on an Intel® Core™ i7-4790K CPU @ 4.00GHz x 4 (8 Threads) on 64-bit with 32 GB RAM and a Samsung SSD 850, running Ubuntu 17.10 with the Kernel Linux 4.13.0-16-generic. All unnecessary services (including graphical user interfaces) were disabled in order not to distort the experiments.

5.1 Functional Evaluation

In this section we are going to evaluate AntTracks’s applicability to detect memory leaks and their root causes. The analyzed application, *mult-cache*, is an artificial demo application to demonstrate AntTracks’s ability to detect memory leaks caused by multiple objects, a situation in which dominator-tree-based approaches prove less useful. It stores products in a database which can be identified by a long `id` as well as a `String name`. Once a `Product` instance is queried, it gets stored in two caches (`HashMap`), one to access products via their `id` and one to access them via their name. Both `HashMaps` are stored in static fields, one in the class `IdCache` and

¹<http://dacapobench.org/> (last accessed May 11, 2018)

²Info: <http://www.scalabench.org/> (last accessed May 11, 2018)

³<https://www.spec.org/jvm2008/> (last accessed May 11, 2018)

Class Name	Retained Heap	Percentage
sun.misc.Launcher\$AppClassLoader @ 0x5dc80ccd8	272,777,184	99.91%
java.util.Vector @ 0x5dc810a28	272,766,416	99.91%
java.lang.Object[10] @ 0x5dc810a48	272,766,384	99.91%
class jku.anttracks.example.data.IdCache @ 0x5d11bfd58	64,385,632	23.58%
java.util.HashMap @ 0x5dc8cd7c8	64,385,624	23.58%
java.util.HashMap\$Node[2097152] @ 0x5d6fc88f8	64,385,576	23.58%
java.util.HashMap\$Node @ 0x5d371c7c8	56	0.00%
java.lang.Long @ 0x5d371c7e8 594072	24	0.00%
Total: 1 of 1,000,000 entries; 999,999 more		
class jku.anttracks.example.data.NameCache @ 0x5dc810a80	40,388,680	14.79%
jku.anttracks.example.data.Product @ 0x5d387dde0	96	0.00%
Total: 3 of 2,000,003 entries; 2,000,000 more		

Figure 10: Dominator tree view in MAT.

one in the class NameCache. To simulate a memory leak, the caches are never cleared and thus grow over time. We compare AntTracks to the Eclipse Memory Analyzer (MAT), a state-of-the-art memory monitoring tool that employs dominator-tree-based memory leak detection. This comparison should demonstrate how tools that only perform dominator-tree-based analyses are not able to derive the root cause of a memory leak that is caused by multiple objects, i.e., by multi-object ownership.

As a baseline, we analyzed the application using MAT, which provides a hierarchical visualization of the dominator tree. Figure 10 shows the dominator tree view for the multi-cache demo application. It reports that the complete heap (about 273 MB) is kept alive by the AppClassLoader, which again holds a Vector that holds an Object[], i.e., the loaded classes. The two classes with the largest retained size are the IdCache (about 64 MB) and the NameCache (about 40 MB). Both keep a HashMap alive (only the IdCache is shown in detail in Figure 10), which further keeps its HashMap\$Node instances alive using an array. Yet, these HashMap\$Nodes only dominate their keys, but not their data. Since the dominator tree is only suitable for detecting single-ownership, it is not possible to infer how much and which memory is kept alive by both caches together, or if further data structures are involved in the memory leak. This is also reflected by the fact that the Object[] that keeps the loaded classes alive is the dominator of 1,000,000 Product instances, which is rather useless to infer ownership (this number is not directly visible in Figure 10 but is a part of the very last entry in the figure).

By default, AntTracks classifies all objects by type, which can be seen in Figure 11. The figure shows that the application's 22 HashMaps together keep 99.9% of the heap alive. When classifying the maps by allocation site (see Figure 12), the classification tree displays the retained sizes of the individual hash maps (23.6% for the one allocated in IdCache, 14.8% for the one allocated in NameCache, and 0.0% for the remaining 20 maps). The individual retained sizes do not add up to the combined retained size of 99.9%, which indicates shared ownership between the hash maps. To analyze the shared ownership of the two caches, both have to be combined into one object group, which in turn may then be analyzed. There are two options to do this: (1) Selecting the two rows in the tree and opening the object group inspection window (showing the object group information explained in Section 4.4 for the two maps combined) or (2) applying a classifier that groups the two caches into one node in the classification tree.

Figure 13 shows parts of the object group inspection window that gets displayed when using the first method. The upper part

Name	Objects		Retained Size [bytes]	
	#	%	#	%
Overall	8,005,995	100.0	273,134,776	100.0
java.util.HashMap	22	0.0	272,776,624	99.9
java.util.HashMap\$Node[]	16	0.0	272,775,840	99.9
java.util.HashMap\$Node	2,000,037	25.0	255,997,488	93.7
jku.anttracks.example.data.Product	1,000,000	12.5	96,000,000	35.1
java.lang.String	1,001,163	12.5	72,097,816	26.4
char[]	1,001,184	12.5	48,104,536	17.6
int[]	2,000,450	25.0	48,029,688	17.6
java.lang.Long	1,000,129	12.5	24,003,096	8.8

Figure 11: AntTracks's default heap state analysis view, classifying heap objects by type.

Name	Objects		Shallow Size [bytes]		Retained Size [bytes]	
	#	%	#	%	#	%
Overall	8,005,995	100.0	273,134,776	100.0	273,134,776	100.0
java.util.HashMap	22	0.0	1,056	0.0	272,776,624	99.9
jku.anttracks.example.data.IdCache	10	0.0	480	0.0	64,385,624	23.6
jku.anttracks.example.data.NameCache	10	0.0	480	0.0	40,388,672	14.8
sun.misc.URLClassPath	20	0.0	960	0.0	3,056	0.0

Figure 12: Classifying heap objects by type and allocation site in AntTracks.

Metric	Value
Shallow size [objects]	2
Shallow size [bytes]	96
Deep size [objects]	8,000,004
Deep size [bytes]	272,769,352
Retained size [objects]	7,999,877
Retained size [bytes]	272,766,304

Name	Objects		Shallow size [bytes]		Retained size [bytes]	
	#	%	#	%	#	%
Overall	7,999,877	100.0	272,766,304	100.0	272,766,304	100.0
java.util.HashMap\$Node	2,000,000	25.0	64,000,000	23.5	255,988,960	93.8
int[]	2,000,000	25.0	48,000,000	17.6	48,000,000	17.6
jku.anttracks.example.data.Product	1,000,000	12.5	48,000,000	17.6	96,000,000	35.2
char[]	1,000,000	12.5	47,992,008	17.6	47,992,008	17.6
java.lang.String	1,000,000	12.5	24,000,000	8.8	71,992,008	26.4
java.lang.Long	999,873	12.5	23,996,952	8.8	23,996,952	8.8
java.util.HashMap\$Node[]	20	0.0	16,777,248	6.2	272,766,208	100.0
java.util.HashMap	20	0.0	960	0.0	272,766,304	100.0

Figure 13: Parts of the object group inspection window displaying information about the hash maps allocated in IdCache and NameCache

of the figure shows various metrics, including the retained size. A retained size of about 272MB confirms our assumption that both caches together nearly keep the whole heap alive. In the lower part, the classification tree (by default classifying objects by their types) of the retained objects is shown. It reveals that, beside other objects, 1,000,000 Product instances are part of the caches' shared GC closure.

When using the second method to analyze shared ownership, a suitable classifier has to be used to group the suspicious objects into one tree node. Since both caches are allocated in the same package, the *allocating package classifier* seems advisable to group them. Additionally, the type classifier, followed by the indirectly-GC-rooted classifier has been applied, and the result can be seen in Figure 14. We can see that 3,000,002 objects have been allocated in the `jku.anttracks.example.data` package, and all objects together have a retained size of 99.9% of all memory. By classifying these objects by type, we can see that they split up into 1,000,000 Product instances, 2,000,000 int[] instances, and 2 HashMaps, i.e., the maps in the caches. The maps also have a retained size of 99.9%, which clearly identifies the two HashMaps as the source

Name	Objects		Retained Size [bytes]	
	#	%	#	%
Overall	8,003,083	100.0	272,994,128	100.0
java	5,001,706	62.5	272,952,440	100.0
jku	3,000,002	37.5	272,766,304	99.9
antracks	3,000,002	37.5	272,766,304	99.9
example	3,000,002	37.5	272,766,304	99.9
data	3,000,002	37.5	272,766,304	99.9
java.util.HashMap	20.0	0.0	272,766,304	99.9
Root: Local variable	20.0	0.0	272,766,304	99.9
Root: Static field	20.0	0.0	272,766,304	99.9
Class: jku.antracks.example.data.IdCache	10.0	0.0	64,385,624	23.6
Field: java.util.Map idCache	10.0	0.0	64,385,624	23.6
Class: jku.antracks.example.data.NameCache	10.0	0.0	40,388,672	14.8
Field: java.util.Map nameCache	10.0	0.0	40,388,672	14.8
jku.antracks.example.data.Product	1,000,000	12.5	96,000,000	35.2
int[]	2,000,000	25.0	48,000,000	17.6
sun	1,375	0.0	113,360	0.0

Figure 14: Classifying heap objects by allocating package, type and indirect GC roots in AntTracks.

GC Roots				
Direct GC Roots				
Indirect GC Roots				
Name	Objects	Shallow size [bytes]	Retained size [bytes]	
#	%	#	%	#
Overall	2,100.0	96,100.0	272,766,304	100.0
Root: Static field	2,100.0	96,100.0	272,766,304	100.0
Class: jku.antracks.example.data.IdCache	1,500.0	48,500.0	64,385,624	23.6
Field: java.util.Map idCache	1,500.0	48,500.0	64,385,624	23.6
Class: jku.antracks.example.data.NameCache	1,500.0	48,500.0	40,388,672	14.8
Field: java.util.Map nameCache	1,500.0	48,500.0	40,388,672	14.8

Figure 15: AntTracks’s object group inspection window also allows the analysis of GC roots.

of the memory leak. To find out which GC roots keep certain objects alive, one can apply the indirectly-GC-rooted classifier. In the case of the two maps this classifier returns two static fields, one in the IdCache class of type `java.util.Map` called `idCache`, and one in the NameCache class of type `java.util.Map` called `nameCache`. Even if the user would not have had any prior knowledge about the system under investigation, we claim that this information is enough to find the corresponding locations in the source code to investigate the leak further on source code level.

This memory leak could have also been identified by using bottom-up analysis. By comparing multiple heap states regarding the frequency of types, one could detect that the number of Product instances is increasing over time. In Section 3.4, such an object group has been called *suspicious object group*. By opening the object group inspection window for such an object group, we can inspect the closures of the objects as well as the GC roots. The hierarchical view of the GC roots also allows us to define classifiers on them. By default, the root objects are classified using the directly-GC-rooted classifier, categorizing them based on their referencing GC roots (see Figure 15). This again identifies the two HashMaps as responsible for the memory leak.

5.2 Recording Overhead

In Section 3.1 we presented the events that are recorded by the AntTracks VM to collect information about GC roots. Information is recorded for every GC root at the start of every garbage collection. Therefore, the recording of this information may have an influence on the following metrics:

- *Run time*: The time it takes to execute a given benchmark.
- *Trace file size*: The size of the resulting trace file.

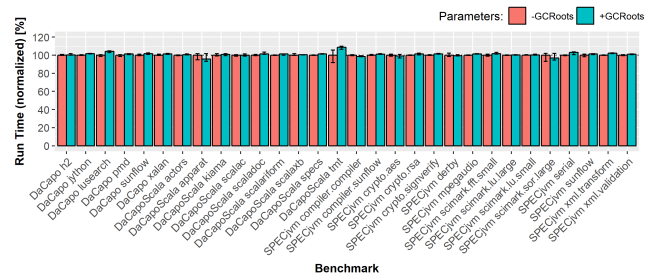


Figure 16: Median run time without (-GCRoots) and with (+GCRoot) GC root tracing, relative to the median run time without GC root tracing enabled.

The recording does not influence the number of garbage collections since we did not modify the GC’s collection behavior, but only perform additional operations at the beginning of each garbage collection.

We evaluate the overhead using benchmarks from the DaCapo suite, the DaCapo Scala suite and the SPECjvm2008 suite, all fixed to a maximum heap size of 2GB (which is enough to run the benchmark with the largest live set, i.e., DaCapo h2) and using the Parallel Old GC. We reduced our selection to benchmarks that trigger at least one garbage collection per run.

As a baseline, we used the AntTracks VM to trace the applications, including pointer information but disabling GC root information tracing (parameter `-GCRoots`). According to Lengauer et al. [21], this introduces an average run time overhead of 15.0%. We measured the additional overhead introduced by enabling GC root information tracing (parameter `+GCRoots`). For warm-up, we ran a benchmark with a given parameter 10 to 40 times on a single VM instance using the largest input size to ensure stabilization of caching and JIT compilation. The largest possible input sizes and the necessary warm-up iterations depend on the benchmarks and have been taken from Lengauer et al., Figure 1 [24]. After warm-up, we ran the benchmark another 10 times on the same VM and calculated the median run time and median trace file size of these runs. We repeated this experiment 10 times for every benchmark and parameter combination, using a new VM instance every time. In the next sections, we report the median, the 25 percentile, and the 75 percentile of the 10 medians per benchmark and parameter setting.

5.2.1 Run time. Figure 16 shows the median run time for each benchmark, relative to the median run time without GC root tracing enabled. The error bars show the 25 percentile and the 75 percentile. On average (geometric mean), the run time increases by 1.00% across all benchmarks when turning GC root tracing on, with an outlier of a maximum median run time increase by 8.77% on the DaCapoScala `tmt` benchmark. The reason for this outlier may be that `tmt` is one of the most allocation-intensive benchmarks. The official documentation describes `tmt` as *externally single-threaded and internally multi-threaded. It creates a large number of threads, each of which is only very short-lived*.⁴ Allocating a lot of objects may trigger many garbage collections, which, in combination with a large number of thread-local GC roots, may lead to a run-time

⁴<http://www.benchmarks.scalabench.org/modules/tmt-dacapo-benchmark/> (last accessed May 11, 2018)

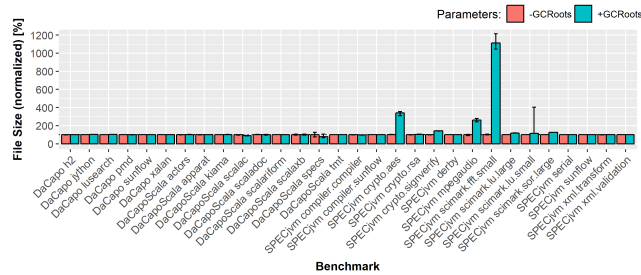


Figure 17: Trace file size without (-GCRoots) and with (+GCRoot) GC root tracing, relative to the median trace file size without GC root tracing enabled.

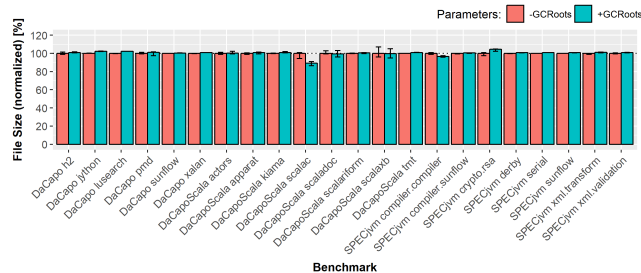


Figure 18: Trace file size without (-GCRoots) and with (+GCRoot) GC root tracing, relative to the median trace file size without GC root tracing enabled (excluding benchmarks generating trace files < 30MB).

increase. Nevertheless, tmt was the only outlier, and we claim that an average run-time increase of one percent makes our approach feasible for production systems.

5.2.2 Trace file size. Figure 17 shows the median trace file size for each benchmark, relative to the median trace file size without GC root tracing. The error bars show the 25 percentile and the 75 percentile. The extreme trace file size increases for certain benchmarks (over 1000% for SPECjvm scimark.fft.small) have to be set in relation to their absolute file sizes. For most benchmarks, the trace files have sizes between some 100 megabytes and some gigabytes, depending on how many objects are allocated by the application and moved by the garbage collector. All benchmarks that show a large relative increase of their trace sizes have an original trace size of less than 30MB. These benchmarks allocate and move very few objects, and the size of the GC root information exceeds the amount of the trace data for the actual objects.

Figure 18 shows the same data as Figure 17, but excludes all benchmarks that generate trace files smaller than 30MB. On average (geometric mean), the trace file size for these benchmarks increases by 0.21% across all benchmarks when turning GC root tracing on. Since most of the data that is written to the trace files concerns object allocations, object movements and pointer updates, the additional data written due to GC roots becomes nearly negligible.

6 RELATED WORK AND STATE-OF-THE-ART

Based on the dominator tree algorithm proposed by Lengauer and Tarjan [41], various variations have been presented [2, 7, 33]. Dominator trees are used in a wide range of applications. For example,

dominance is used in compilers to analyze control flow graphs [10] or to visualize software dependencies [14].

In the domain of program understanding and memory leak detection, dominator trees (alongside other techniques) are often used as basis for ownership detection, component analysis and aggregation in heap graph visualization. Hill et al. [17, 18] reduce object graphs into *ownership trees* for visualization based on the dominator tree. Rayside et al. [34, 35] introduce *object ownership profiling*, a technique that uses the dominator tree alongside information about last object access times and object interactions to identify memory leaks and memory management anti-patterns in applications. Mitchell [27] summarizes an application’s memory footprint with help from the dominator relation. He introduces a set of ownership structures, detects these structures in object graphs and aggregates them into concise *ownership graphs* that visualize responsibility and ownership of data structures. In further work, Mitchell and Sevitsky [29] introduce *health signatures* for data structures based on dominator tree reference analysis. They show how to judge data structure designs or implementations based on their relationship between actual data and structural overhead. In [28], Mitchell et al. present aggregation techniques using the domination relation in heap object graphs to perform progressive graph abstractions, alongside corresponding visualizations. Similarly, other approaches use dominator information to abstract object graph visualizations [26, 36–38] or to layout graphs [1].

Nevertheless, ample work in the domain of memory leak detection exists that does not rely on dominator trees but uses object graphs directly, as we do in AntTracks. Most of these approaches rely on certain types of pattern detection in the object graphs. De Pauw et al. [11, 12] extract patterns from object graphs, shifting the focus from individual objects to groups of objects to abstract their visualization. Jump and McKinley [19] developed the memory leak detection tool *Cork* that reduces the object graph to a *type-points-to-type* graph for analysis. Barr et al. [3] use the object graph to detect and classify typical reference patterns in real-world application heaps. Chis et al. [6] discuss the limitations of the dominator relation (mostly due to the issue of detecting shared ownership), alternatively describe a *ContainerOrContained* relation, and use this relation to detect various inefficient memory patterns.

State-of-the-art memory monitoring tools share the typical functionality to represent heap states as type histograms, showing the number of instances per class and their shallow sizes. While this enables users to detect large object groups of a certain type in the same way in all tools, they differ in how they support memory leak root cause detection.

VisualVM [32] is a general performance monitoring tool for Java applications that provides memory analysis based on heap dumps. In addition to a type histogram, it can display a list of all root objects, as well as the dominator tree. From each view, it is possible to inspect individual objects, including functionality to inspect an object’s fields, accessing referencing objects, and finding the closest root object. Even though VisualVM can calculate the retained size for the objects of a given type, it is not possible to change that classification or to further split that group. Neither can the user select multiple objects that might have shared ownership for inspection, a functionality that is available in AntTracks by using its object group inspection window on arbitrary object groups. The

Netbeans profiler [31] is a slimmed down version of VisualVM and is integrated into the Netbeans IDE.

The focus of the the *Eclipse Memory Analyzer (MAT)* [15] tool is to provide a fast overview on possible memory leaks, reducing the need for complex user interaction. By default, it displays overview charts of the largest dominating objects, packages and class loaders. Their computations heavily rely on the dominator tree. In addition, MAT provides an automatic leak suspect analysis which detects and extracts the most suspicious objects from the dominator tree. While MAT provides easy-to-use automated analysis features and high-level abstractions based on the dominator tree, it shares a common problem with VisualVM: Memory leak root cause detection is not supported for leaking object groups with shared ownership.

7 FUTURE WORK AND THREATS TO VALIDITY

Heap object graphs, alongside closure, could also be reconstructed and calculated using normal heap dumps. Yet, the analysis of a single heap state may not be sufficient to detect and analyze the proliferation of objects. Thus, AntTracks utilizes a trace-based method to continuously record memory information. Trace files can then be used to analyze an application's memory behavior *over time* on *object-level*. Future work will make use of the temporal information that can be reconstructed from traces and will combine it with the approaches presented in this work. This includes automated analysis of changes to the heap object graph, i.e., how object references and GC root pointers change over time. This will enable us to automatically detecting continuously growing object groups, e.g., detecting growing object groups of certain types or allocation sites. It will also be possible to detect objects with a growing retained size, which might hint at growing data structures. We further plan to include a constraint DSL in AntTracks which would enable users to define memory constraints such as maximum retained sizes or checks for invalid reference patterns. Such constraints can then be checked during parsing of trace files.

After detecting growing object groups (e.g., due to a memory leak) or object groups that keep large portions of the heap alive, the user's goal is to make these objects eligible for garbage collection. This can be done by cutting references between objects on the paths to their GC roots. Analyzing these paths to find suitable cutting points is most effectively done by visualizing the object graph. We therefore plan to develop more sophisticated graph visualization techniques that offer convenient navigation and analysis of object graphs. Yet, without aggregation, such object graphs tend to grow big and become infeasible to analyze. Thus, our goal is to reduce this complexity by collapsing the object graph based on data structure membership, i.e., by aggregating objects that belong to a certain data structure into a single graph node. When searching for suitable cutting points, one is not interested in the internals of data structures. For example, a linked list maintains its elements via Node objects. These internals should not show up in the object graph. By aggregating all objects of the list into a single node, one could raise the abstraction level of the analysis. As a first step, data structure aggregation could be done for the well-known Java collection types. Later it could also be done for arbitrary user-defined data structures described by a domain-specific language.

The major threat to validity of our work is its currently restricted evaluation based on an artificial use case. It also lacks an evaluation on how often multi-object ownership occurs in real-world applications. There exist studies on the memory behavior of real-world applications [8, 9, 16], yet they do not evaluate the interaction between objects or data structures, e.g. they do not report multi-object ownership rates. We plan to conduct in-depth evaluations on open-source projects in the future to gain a deeper understanding of object interaction in real-world applications. Further, to prove AntTracks's applicability for finding the root cause of memory leaks in such real-world applications, we also plan to conduct a user study with our industry partner. In addition to comparing AntTracks to existing tools, e.g., in terms of found memory leaks, we also want to evaluate which classifier combinations are most useful when searching for memory leaks.

8 CONCLUSION

In this paper, we presented new techniques for collecting information about GC roots and how to use this information for computing the transitive closure and the GC closure of the object graphs referenced by these roots. A distinguishing feature of our approach is the fact that we can compute the GC size for whole object *groups* and not only for single objects, as is the case in dominator-tree-based approaches. From the closures we derived metrics such as the *retained size* of an object group (i.e., the amount of memory that is kept alive by this group). Finally, we integrated our techniques into a state-of-the-art memory monitoring tool (AntTracks) that provides classification and navigation facilities for analyzing the memory behavior of an application and finding the root causes of memory leaks.

The GC root information is written to a trace file at the start of every garbage collection and can thus be reconstructed offline for any garbage collection point. In a quantitative evaluation based on the DaCapo, DaCapoScala and SPECjvm2008 benchmark suites, we showed that tracing GC root information introduces 1.00% overhead on the application's run time and 0.21% overhead on the generated trace file size on average, which is low enough to be used in production systems.

A functional evaluation showed that our approach for computing the GC closure enables us to detect memory leaks even if the leaking objects are shared by multiple owner objects. In particular, the retained size metric proved useful to detect data structures that have shared ownership. Finally, we showed how bottom-up analysis can be used to find the GC roots that keep a set of leaking objects alive.

ACKNOWLEDGMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria GmbH.

REFERENCES

- [1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proceedings of the 5th International Symposium on Software Visualization (SOFTVIS '10)*. ACM, New York, NY, USA, 53–62. <https://doi.org/10.1145/1879211.1879222>
- [2] Stephen Alstrup and Peter W. Lauridsen. 1996. *A Simple Dynamic Algorithm for Maintaining a Dominator Tree*. Technical Report.

- [3] Earl T Barr, Christian Bird, and Mark Marron. 2013. Collecting a heap of shapes. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 123–133.
- [4] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*. 76–89.
- [5] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.
- [6] Adriana E Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of memory inefficiency. In *European Conference on Object-Oriented Programming*. Springer, 383–407.
- [7] Keith D Cooper, Timothy J Harvey, and Ken Kennedy. 2001. A simple, fast dominance algorithm. *Software Practice & Experience* 4, 1–10 (2001), 1–8.
- [8] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. [n. d.]. Empirical Study of Usage and Performance of Java Collections. ([n. d.]). <https://doi.org/10.1145/3030207.3030221>
- [9] Diego Costa and Rivalino Matias Jr. 2015. Characterization of Dynamic Memory Allocations in Real-World Applications: An Experimental Study. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 93–101. <https://doi.org/10.1109/MASCOTS.2015.28>
- [10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '89)*. ACM, New York, NY, USA, 25–35. <https://doi.org/10.1145/75277.75280>
- [11] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vliissides, and Jaeha Yang. 2002. Visualizing the execution of Java programs. In *Software Visualization*. Springer, 151–162.
- [12] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *ECOOP'99 – Object-Oriented Programming*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 116–134.
- [13] J. Eve and R Kurki-Suonio. 1977. On computing the transitive closure of a relation. *Acta Informatica* 8, 4 (oct 1977), 303–314. <https://doi.org/10.1007/BF00271339>
- [14] R. Falke, R. Klein, R. Koschke, and J. Quante. 2005. The Dominance Tree in Visualizing Software Dependencies. In *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 1–6. <https://doi.org/10.1109/VISSOF.2005.1684311>
- [15] Eclipse Foundation. 2018. Eclipse Memory Analyzer (MAT) (last accessed May 11, 2018). <https://www.eclipse.org/mat/>. (2018).
- [16] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE '18)*. ACM, New York, NY, USA, 410–411. <https://doi.org/10.1145/3183440.3195032>
- [17] T. Hill, J. Noble, and J. Potter. 2000. Scalable visualisations with ownership trees. In *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000*. 202–213. <https://doi.org/10.1109/TOOLS.2000.891370>
- [18] Trent Hill, James Noble, and John Potter. 2002. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages & Computing* 13, 3 (2002), 319–339.
- [19] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*. ACM, New York, NY, USA, 31–38. <https://doi.org/10.1145/1190216.1190224>
- [20] Donald E. Knuth. 1971. Top-down syntax analysis. *Acta Informatica* 1, 2 (01 Jun 1971), 79–110. <https://doi.org/10.1007/BF00289517>
- [21] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.
- [22] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conference on Performance Engineering (ICPE '15)*. 51–62.
- [23] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '16)*. 249–260.
- [24] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 3–14. <https://doi.org/10.1145/3030207.3030211>
- [25] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (Jan. 1979), 121–141. <https://doi.org/10.1145/357062.357071>
- [26] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. 2013. Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering* 39, 6 (June 2013), 774–786. <https://doi.org/10.1109/TSE.2012.69>
- [27] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 74–98. https://doi.org/10.1007/11785477_5
- [28] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making Sense of Large Heaps. In *Proceedings of the 23rd European Conference on ECOOP 2009 – Object-Oriented Programming (Genoa)*. Springer-Verlag, Berlin, Heidelberg, 77–97. https://doi.org/10.1007/978-3-642-03013-0_5
- [29] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications (OOPSLA '07)*. ACM, New York, NY, USA, 245–260. <https://doi.org/10.1145/1297027.1297046>
- [30] Oracle. 2018. The HotSpot Group (last accessed May 11, 2018). <http://openjdk.java.net/groups/hotspot/>. (2018).
- [31] Oracle. 2018. Netbeans Profiler (last accessed May 11, 2018). <https://profiler.netbeans.org/>. (2018).
- [32] Oracle. 2018. VisualVM: All-in-One Java Troubleshooting Tool (last accessed May 11, 2018). <https://visualvm.github.io/>. (2018).
- [33] G. Ramalingam and Thomas Reps. 1994. An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '94)*. ACM, New York, NY, USA, 287–296. <https://doi.org/10.1145/174675.177905>
- [34] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*. ACM, New York, NY, USA, 194–203. <https://doi.org/10.1145/1321631.1321661>
- [35] Derek Rayside, Lucy Mendel, and Daniel Jackson. 2006. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis (WODA '06)*. ACM, New York, NY, USA, 57–64. <https://doi.org/10.1145/1138912.1138924>
- [36] Steven P Reiss. 2009. Visualizing the Java heap - Demonstration Proposal. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 389–390.
- [37] S. P. Reiss. 2009. Visualizing the Java heap to detect memory problems. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 73–80. <https://doi.org/10.1109/VISSOF.2009.5336418>
- [38] Steven P. Reiss. 2010. Visualizing the Java Heap. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 251–254. <https://doi.org/10.1145/1810295.1810344>
- [39] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise Gc Traces. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM '13)*. ACM, New York, NY, USA, 109–118. <https://doi.org/10.1145/2464157.2466484>
- [40] C. Ruggieri and T. P. Murtagh. 1988. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. ACM, New York, NY, USA, 285–293. <https://doi.org/10.1145/73560.73585>
- [41] Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*. IEEE, 114–121. <https://doi.org/10.1109/SWAT.1971.10>
- [42] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l. Conference on Performance Engineering (ICPE '17)*. 357–360.
- [43] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 9th ACM/SPEC International Conference on Performance Engineering (ICPE 2018) (ICPE 2018)*.

Chapter 4

Data Structure Analysis

This chapter includes two papers [316, 318] that describe our automatic data structure growth analysis approach that tracks data structures across their lifetime and sorts them based on their growth behavior and possible involvement in a memory leak.

Work-In-Progress Paper:

Markus Weninger, Elias Gander, Hanspeter Mössenböck:

Analyzing the Evolution of Data Structures in Trace-Based Memory Monitoring. In *Proceedings of the 9th Symposium on Software Performance, SSP 2018*, Hildesheim, Germany, November 8 - 9, 2018.

Full Paper:

Markus Weninger, Elias Gander, Hanspeter Mössenböck:

Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019*, Mumbai, India, April 7-11, 2019.

Analyzing the Evolution of Data Structures in Trace-Based Memory Monitoring

Markus Weninger^{*⊗}, Elias Gander[⊗], Hanspeter Mössenböck^{*}
{`firstname.lastname@jku.at`}

^{*} Institute for System Software, Johannes Kepler University, Linz

[⊗] Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz

Abstract

Modern software systems are becoming increasingly complex and are thus more prone to performance degradation due to memory leaks. Memory leaks occur if objects that are not needed anymore are still unintentionally kept alive. While there exists a variety of state-of-the-art memory monitoring tools, most of them only use memory snapshots, i.e., heap dumps, to analyze an application’s live objects at a single point in time. This does not allow developers to identify data structures that grow over time. Trace-based monitoring tools tackle this problem by recording memory events, e.g., allocations or object moves performed by the garbage collector (GC), throughout an application’s run time. In this paper, we present ongoing research on the use of memory traces for detecting the root causes of memory leaks introduced by growing data structures. This encompasses (1) a domain-specific language (DSL) to describe arbitrary data structures, (2) an algorithm to detect instances of previously defined data structures in reconstructed heaps, as well as (3) techniques to analyze the temporal evolution of these data structure instances to identify those possibly involved in memory leaks. All these concepts have been integrated into AntTracks, a trace-based memory monitoring tool, to prove their feasibility.

1 Introduction

Modern programming languages such as Java use automatic garbage collection. During garbage collection, objects that are not directly or indirectly reachable from static fields or thread-local variables (so-called *GC roots*) may be collected by the GC. We speak of a memory leak if no longer needed objects remain reachable from GC roots due to a programming error. For example, if a developer misses to remove no longer needed objects from their containing data structures, e.g., lists, these objects may not be collected by the GC. Beside excessive dynamic allocations [1], memory leaks are one of the major memory anomalies [4].

Since modern applications may involve hundreds of millions of objects at a single point in time, tool support to resolve memory problems is of paramount importance. Most state-of-the-art tools, such as Vi-

sualVM [8] or Eclipse Memory Analyzer (MAT) [7], perform heap analysis based on snapshots, i.e., heap dumps. While a single heap dump may allow developers to detect large data structures, it provides no information about the heap’s evolution over time. Thus, some approaches [2, 8] take multiple snapshots and compare them. Nevertheless, such approaches do not allow temporal analyses on the *object-level*.

In contrast to snapshot-based approaches, trace-based approaches record additional information, e.g., object moves executed by the GC. This allows them to reconstruct the heap offline from the recorded trace for any point in time, as well as to track specific objects and their evolution throughout an application. Since it would not be feasible – due to memory restrictions and computational borders – to reconstruct and remember every single change to every single object, temporal analysis approaches need to focus on a certain subset of objects.

In this work, we present ongoing research on the use of memory traces. Our goal is to extract information about the root causes of memory leaks by focusing on the temporal development of *data structures*. This encompasses (1) a DSL that enables users to describe arbitrary data structures, (2) an algorithm to detect instances of previously defined data structures in reconstructed heaps, as well as (3) techniques to analyze the temporal evolution of these data structure instances to identify those possibly involved in memory leaks. To prove the feasibility of our approach, all concepts have been integrated into AntTracks. AntTracks is a trace-based memory monitoring tool based on the Hotspot Java VM, initially developed by Lengauer et al. [3] and extended by Weninger et al. [5, 6].

2 Approach

This section illustrates how data structures can be described in our DSL, how they are detected in reconstructed heaps, and how information about their temporal evolution is derived from AntTracks’s memory traces.

2.1 Data Structure Definition

In object-oriented languages such as Java, data structures typically consist of a *head* object and multiple

other objects that reference each other according to a specific pattern. These patterns have to be known by a memory monitoring tool in order to enable it to perform data structure analyses. Therefore, we developed a DSL that allows us to describe arbitrary data structures. This allows us to ship descriptions of well-known data structures (e.g., data structures in Java’s `java.util` package) directly with AntTracks. At the same time, tool users can extend this set of predefined data structure descriptions with descriptions of their own data structures.

Listing 1 shows an example of the DSL, describing the structure of `java.util.LinkedList`. Every type that is involved in the data structure needs a description, i.e., in our example `java.util.LinkedList` and `java.util.LinkedList$Node`. The former represents the *head* of the data structure (marked with the DS keyword), while the latter is an internal part of a data structure. Similar to Java syntax, the name of the type is followed by a set of curly braces. These contain a set of types (separated by semicolons) that may be referenced by the respective data structure part. For example, an instance of `java.util.LinkedList` may point to instances of `java.util.LinkedList$Node` (see line 2), which in turn may point to instances of `java.util.LinkedList$Node` instances (see line 5), and so on. Line 6 presents two special language features: (1) a star (i.e., `*`) can be used as a wildcard within the name of a pointed type and (2) enclosing a type in parentheses declares it as a *leaf*. The term `(*)` denotes a leaf of any type. Leaf information is used during data structure detection to determine the boundaries of a data structure. The DSL also supports namespaces which allow us to omit package declarations in type names.

2.2 Data Structure Detection

In order to detect data structures in a heap, the data structure definitions have to be parsed first. The parsed definitions are assigned to their corresponding types. Types for which no data structure definition was parsed are assigned a non-head dummy data structure definition that does not declare any pointed types. Array types are an exception to this rule and are handled in a special way. At this point, every type has a data structure definition assigned.

A reconstructed heap contains information about the objects live at a certain point in time (e.g., their types), as well as their references between each other. As a first step, the data structure detection algorithm filters and remembers all objects that are data structure heads, i.e., objects whose types have a *head* data structure definition assigned. Then, to determine which objects belong to a certain data structure instance, it recursively follows the head object’s pointers. The recursive descent is stopped when a pointed object is encountered whose type is either (1) not part of the current object’s data structure definition or (2)

```

1 DS java.util.LinkedList {
2   java.util.LinkedList$Node;
3 }
4 java.util.LinkedList$Node {
5   java.util.LinkedList$Node;
6   (*);
7 }

```

Listing 1: Definition of `java.util.LinkedList` using our data structure DSL.

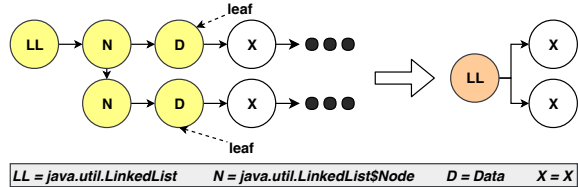


Figure 1: A `LinkedList` instance, consisting of the head (LL), two nodes (N) and two data objects (D).

marked as a leaf in the current object’s data structure definition. In the latter case, the object itself belongs to the data structure instance, but none of its referenced objects. Every visited object is marked to avoid multiple visits.

For example, Figure 1 shows a `java.util.LinkedList` that has been detected using the description in Listing 1. Starting at the head LL, the first N instance is visited. The data structure description of `java.util.LinkedList$Node` then allows us to follow further nodes (line 5), or to visit any other object as a leaf without continuing the recursive descent (line 6). Thus, the first D instance and the second N instance are visited, continuing the descent from the N object. As a last step, the second D object is visited as a leaf.

2.3 Temporal Analysis

Trace-based approaches are better suited for temporal analysis than snapshot-based ones because they allow to derive temporal information on the *object-level*. Figure 2 illustrates this. Using only snapshots, without additional temporal information, it is not possible to decide whether two objects of type X are really the same or just share the same type.

AntTracks is able to derive this information by replaying the recorded GC move events. Thus, we know which objects survived between two points in time as well as their updated pointers. Using this knowledge, we can specifically search for data structure instances which (1) survived over a certain time window (since

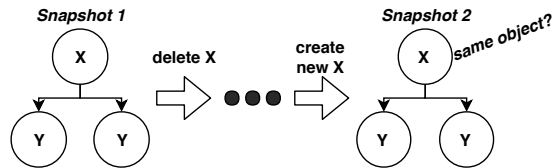


Figure 2: Analysis based on multiple snapshots lacks information *on the object-level*.

objects that died cannot be the root cause of a memory leak) and (2) reference / keep alive more objects than before.

Our workflow for temporal data structure evolution analysis consists of the following steps:

1. The user chooses two garbage collection points in time between which the temporal data structure evolution analysis should take place.
2. The heap is reconstructed for the first point in time and is stored. The addresses of all data structure heads in this heap, i.e., the *start addresses*, are stored as well.
3. At every garbage collection, we stop tracking data structure heads that died. For surviving heads, their new addresses (which can be reconstructed from GC move events) are stored alongside their start addresses.

Following this algorithm up to the end of the selected time window, we obtain (1) the reconstructed heap at the start, (2) the reconstructed heap at the end, and (3) a list of all data structures that survived, more specifically, their initial and final addresses.

For every data structure head, its *deep size* (i.e., how many objects can be reached from that object) as well as its *retained size* (i.e., how many objects are kept alive by that object) can be calculated for both points in time [5]. Based on that, the absolute and the relative change of these sizes can be calculated.

The metric that proved the most useful to identify problematic data structure instances in our preliminary evaluation is shown in Equation 1.

$$HGP(obj) = \frac{\Delta retained(obj)}{\Delta heap size} \times 100 \quad (1)$$

Given that the overall heap size increased, this formula calculates the ownership growth of each data structure relative to the heap growth, i.e., the *heap growth portion*. For example, assume that the overall heap size went from 1GB to 2GB and a list’s retained size increased by 700MB. This would result in a *HGP* value of 70%, i.e., the ownership growth of this data structure contributes 70% to the total growth of the heap.

Sorting all data structures by this metric allows us to easily identify those that keep more objects alive than before. At the same time their growth is put into perspective to the absolute heap growth. In the case of a memory leak, objects that reveal a high *HGP* value are most likely involved in it.

3 Conclusion and Future Work

In this paper, we presented a new and easy-to-use DSL to describe arbitrary data structures and sketched an algorithm that detects instances of those data structures in reconstructed heaps. We discussed how temporal information regarding the growth of data structure instances can be derived from memory traces,

including a metric that puts data structure growth in relation to the overall heap growth. This metric allows us to prioritize data structure instances according to how likely they are involved in a memory leak.

Being able to describe, detect and analyze the evolution of arbitrary data structures over time, even user-defined ones, yields many possibilities for future work. Due to the complexity of heap object graphs, it is not feasible to visualize and inspect them without abstraction, e.g., by aggregating nodes. Our work can be used to develop improved object graph visualization techniques that perform node aggregation based on data structure information. Data structure information may also be used to push automatic memory leak detection and resolution without human intervention.

4 Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

- [1] C. U. Smith and L. G. Williams. “Software performance antipatterns.” In: *Workshop on Software and Performance*. 2000.
- [2] M. Jump and K. S. McKinley. “Detecting memory leaks in managed languages with Cork”. In: *Software: Practice and Experience* 40.1 (2010).
- [3] P. Lengauer, V. Bitto, and H. Mössenböck. “Accurate and Efficient Object Tracing for Java Applications”. In: *Proc. of the 6th ACM/SPEC Int’l. Conference on Performance Engineering*. 2015.
- [4] M. Ghanavati et al. “Memory and Resource Leak Defects in Java Projects: An Empirical Study”. In: *Proc. of the 40th Int’l Conf. on Software Engineering: Companion Proceedings*. 2018.
- [5] M. Weninger, E. Gander, and H. Mössenböck. “Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring”. In: *Proc. of the 15th Int’l Conf. on Managed Languages & Runtimes*. 2018.
- [6] M. Weninger and H. Mössenböck. “User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring”. In: *Proc. of the 9th ACM/SPEC Int’l Conf. on Performance Engineering*. 2018.
- [7] Eclipse Foundation. *Eclipse Memory Analyzer (MAT) (last accessed August 13, 2018)*. <https://www.eclipse.org/mat/>.
- [8] Oracle. *VisualVM: All-in-One Java Troubleshooting Tool (last accessed August 13, 2018)*. <https://visualvm.github.io/>.

Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection

Markus Weninger
Institute for System Software,
CD Laboratory MEVSS,
Johannes Kepler University
Linz, Austria
markus.weninger@jku.at

Elias Gander
CD Laboratory MEVSS,
Johannes Kepler University
Linz, Austria
elias.gander@jku.at

Hanspeter Mössenböck
Institute for System Software,
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

ABSTRACT

Memory leaks are a major threat in modern software systems. They occur if objects are unintentionally kept alive longer than necessary and are often indicated by continuously growing data structures.

While there are various state-of-the-art memory monitoring tools, most of them share two critical shortcomings: (1) They have no knowledge about the monitored application’s data structures and (2) they support no or only rudimentary analysis of the application’s data structures over time.

This paper encompasses novel techniques to tackle both of these drawbacks. It presents a domain-specific language (DSL) that allows users to describe arbitrary data structures, as well as an algorithm to detect instances of these data structures in reconstructed heaps. In addition, we propose techniques and metrics to analyze and measure the evolution of data structure instances over time. This allows us to identify those instances that are most likely involved in a memory leak. These concepts have been integrated into AntTracks, a trace-based memory monitoring tool. We present our approach to detect memory leaks in several real-world applications, showing its applicability and feasibility.

CCS CONCEPTS

• **General and reference** → **Metrics; Performance**; • **Information systems** → **Data structures**; • **Software and its engineering** → **Data types and structures; Domain specific languages; Dynamic analysis; Software performance**; Garbage collection.

KEYWORDS

Memory Monitoring, Data Structures, Growth Analysis, Analysis Over Time, Memory Leak Detection, Domain Specific Language

ACM Reference Format:

Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE ’19)*, April 7–11, 2019, Mumbai, India. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3297663.3310297>

ICPE ’19, April 7–11, 2019, Mumbai, India

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author’s version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Tenth ACM/SPEC International Conference on Performance Engineering (ICPE ’19)*, April 7–11, 2019, Mumbai, India, <https://doi.org/10.1145/3297663.3310297>.

1 INTRODUCTION

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). A memory leak occurs if objects that are no longer needed remain reachable from GC roots due to programming errors. For example, a developer may forget to remove objects from their containing data structures. These objects cannot be reclaimed by the garbage collector and will therefore accumulate over time. Beside excessive dynamic allocations [27–29], memory leaks are one of the major memory anomalies [10].

Applications may involve hundreds of millions of objects at a single point in time. Thus, tools to resolve memory problems are of paramount importance. Most state-of-the-art tools, such as VisualVM [34] or Eclipse Memory Analyzer (MAT) [32], perform heap analyses based on snapshots, i.e., heap dumps. While such tools can group heap objects by their types, they have no notion on how these objects are connected as data structures. This is problematic because memory leaks are frequently related to data structures [41]. By recognizing data structures, users can be provided with further guidance during memory leak detection.

In addition to the problem of missing data structure information, a single heap dump does not give any insights regarding the heap’s evolution over time. Thus, some approaches [6, 12, 13, 34], take multiple snapshots and compare them. Nevertheless, this still does not allow temporal analyses on the *object-level*, i.e., its not possible to tell whether a certain object was alive in snapshot *A* and is still alive in a later snapshot *B*.

In contrast to snapshot-based approaches, trace-based approaches continuously record information about events, e.g., allocations or object moves executed by the GC, throughout an application’s life time. The recorded trace can later be used to reconstruct the heap for an arbitrary garbage collection point. In addition to that, detailed trace-based approaches are able to track specific objects over multiple garbage collections. One example of a trace-based memory monitoring tool is AntTracks, which is based on the Hotspot Java VM. It was initially developed by Lengauer et al. [17] and has been extended by Weninger et al. [37–39]. All concepts presented in this work have been integrated into AntTracks to prove the feasibility of our approach.

Weninger et al. [36] presented first ideas on how to use memory traces to find the root causes of memory leaks by focusing on the *growth of data structures over time*. In this work, we extend and complement our work by a more in-depth description of the approach and the algorithms used, new metrics and metric patterns

for data structure growth analysis, as well as a thorough evaluation of our implementation based on several real-world scenarios in which we detect memory leaks caused by growing data structures.

Our scientific contributions are

- (1) a DSL that enables users to describe arbitrary data structures,
- (2) an algorithm to detect instances of previously defined data structures in reconstructed heaps,
- (3) techniques, metrics and patterns for data structure growth analysis to identify data structures that are possibly involved in memory leaks, as well as
- (4) an evaluation of our approach based on memory leak detection in real-world applications.

2 BACKGROUND

AntTracks consists of two parts: The AntTracks VM, a virtual machine based on the Java Hotspot VM [33], and the AntTracks Analyzer, a memory analysis tool. Since the concepts presented in this paper have been integrated into AntTracks, it is essential to understand how AntTracks works.

2.1 Trace Recording by the AntTracks VM

The AntTracks VM records memory events such as object allocation events and object movements executed by the GC by writing them into trace files. It keeps the event size to a minimum and avoids the recording of redundant data [16, 17].

2.2 AntTracks Analyzer

2.2.1 Reconstruction.

The AntTracks Analyzer is able to parse previously created trace files. The events in the trace are incrementally processed, which enables to reconstruct the heap at every garbage collection point [1]. A heap state is the set of heap objects that were live in the monitored application at a certain point in time. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by. To allow addressing a specific object within a heap state, every heap object is assigned a unique index.

2.2.2 Heap Object Classification.

The AntTracks Analyzer’s core mechanism is object classification in combination with multi-level grouping [38, 39] to enable user-driven heap analysis. Using object classifiers, heap objects can be grouped according to certain criteria such as type, allocation site, allocating thread, and so on. For example, the *Type* classifier allows to group objects by their types, e.g. `java.util.LinkedList`. In multi-level grouping, objects are grouped according to the classification results of multiple classifiers. This results in a hierarchical classification tree.

A common classifier combination is to first group all heap objects by their types (using the *Type* classifier) and then by their allocation sites (using the *Allocation Site* classifier). Figure 1 shows an example of a classification tree. Yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object[]`, of which the objects 0, 1 and 3 have

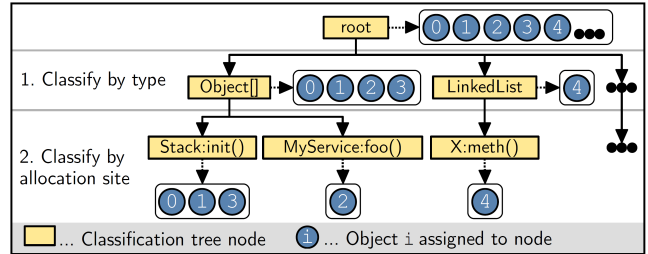


Figure 1: A *classification tree* that first groups all objects by their types and then by their allocation sites.

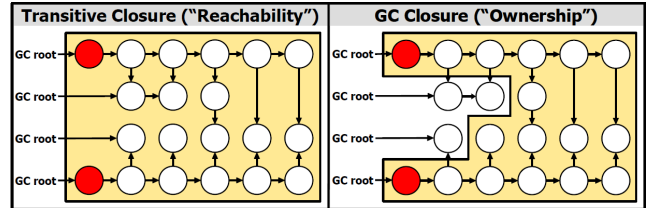


Figure 2: Shared *transitive closure* (“reachability”) and *GC closure* (“ownership”) of the two red objects.

been allocated in `Stack:init()` and object 2 has been allocated in `MyService:foo()`.

2.2.3 Closures and Metrics.

Users require guidance to decide how they should navigate through a classification tree. AntTracks currently supports three metrics that are displayed for every object group, i.e., for every node in the classification tree:

- *Shallow*
The shallow object count and the shallow byte count are calculated based on the objects classified at a given node, without taking into account any referenced objects. For example, the shallow object count of the node `Object[]` in Figure 1 is 4. The shallow byte count is the size of the arrays themselves, without taking into account the sizes of the objects referenced by them.
- *Deep*
The deep object count and the deep byte count are the number of objects / number of bytes of a node’s *transitive closure*. The transitive closure contains all objects that are *reachable* from a given object group, as shown in Figure 2.
- *Retained*
The retained object count and the retained byte count are the number of objects / number of bytes of a node’s *GC closure*. The GC closure contains all objects that are *owned* by a given object group, as shown in Figure 2. In other words, the GC closure contains all objects that could be freed by the garbage collector if the given object group would be freed.

3 APPROACH

Over the last years, the memory consumption of applications has grown drastically. This poses a challenge to memory monitoring tools because it results in more complex heap states that have to be visualized in a user-friendly way. Many tools still use flat type histograms (see Figure 4) as their main visualization.

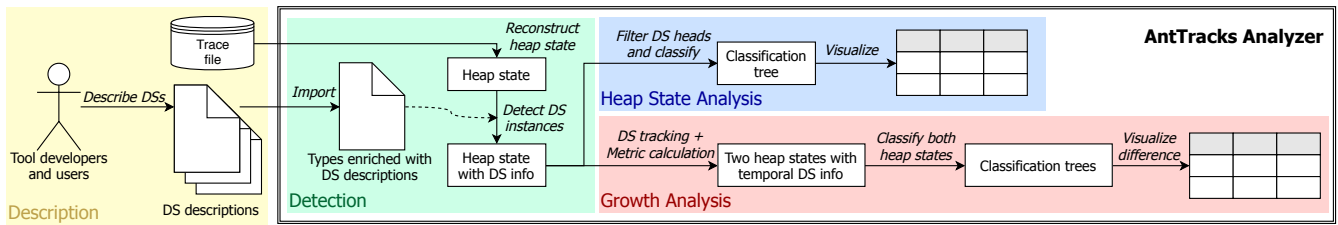


Figure 3: Our approach consists of four stages: (1) Description of data structures (DS) by a DSL, (2) detection of data structure instances in reconstructed heap states, (3) heap state analysis, i.e., data structure analysis at a single point in time, and (4) growth analysis, i.e., tracking data structures over time, detecting those with suspicious growth.

Type	# Objects	Shallow Size
java.util.HashMap\$Node	1.000.000	24 MB
my.package.MyType	800.000	16 MB
int[]	100.00	50 MB
...

Figure 4: A type histogram displays every type alongside the number of allocated objects and the consumed memory.

In many heap states, most of the objects are auxiliary objects, i.e., internal parts of more complex data structures. These are often located at the top of type histograms. One prominent example are `java.util.HashMap$Node` instances. Though head objects of data structures (e.g., of type `HashMap`) are far more likely the root cause for a memory leak, they are only listed at a lower position.

In this work, we present an approach that greatly reduces the complexity that users have to cope with during memory analysis. The idea is to hide objects that convey little information and to focus on the analysis of data structures instead. Generally speaking, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data [35]. For memory leak detection, we are especially interested in the relationships among the objects that make up the data structure.

This section explains the core concepts of our approach (see Figure 3): How data structures can be described by a DSL, how they are detected in a reconstructed heap state, how this information can be used to ease heap state analysis, and how information about data structure growth over time can be derived and used in trace-based tools such as AntTracks.

3.1 Data Structure Description

In object-oriented languages such as Java, data structures typically consist of a *head* object, multiple internal objects that serve as a backbone, and leaf objects that represent the actual contents of the data structure. These objects reference each other according to a specific pattern. This pattern has to be known by a memory monitoring tool before it can detect instances of the respective data structure and perform analyses on it.

3.1.1 Benefits of a DSL for Data Structure Description.

The most straightforward way to achieve pattern recognition would be to hard-code the patterns of well-known data structures directly into the tool’s data structure instance detection algorithm. However, the set of data structure types that can be detected by the tool would be fixed. Users could not define new data structures, for example data structures specific to their application or data structures introduced by third-party libraries. Also, if the well-known data structures change in the future, e.g., due to renamed types, the tool’s source code would have to be modified.

```

1 DS java.util.LinkedList {
2   java.util.LinkedList$Node;
3 }
4 java.util.LinkedList$Node {
5   java.util.LinkedList$Node;
6   (*);
7 }
8
9 namespace java.util {
10  DS LinkedList {
11    *;
12  }
13  LinkedList$Node {
14    LinkedList$Node;
15    (*);
16  }
17 }

```

Figure 5: Description of `java.util.LinkedList` in our DSL, without and with using namespaces and wildcards.

To circumvent these drawbacks, we developed a DSL for describing arbitrary data structures in separate files. These data structure description files can then be read by memory analysis tools such as AntTracks to be used for data structure instance detection in reconstructed heap states. Using a DSL for data structure description instead of hard-coded data structure patterns has various advantages. It enables us to ship descriptions of well-known data structures (e.g., data structures in Java’s `java.util` package) directly with AntTracks. At the same time, tool users can extend the pre-defined data structure descriptions with descriptions of their own data structures. Finally, changes to existing data structures do not require changes in the source code, but only in the data structure description file(s) that are much easier to adjust.

3.1.2 DSL Format.

The left side of Figure 5 shows how `java.util.LinkedList` can be described in our DSL. Every type that is a part of the data structure needs a description, i.e., in our example `java.util.LinkedList` and `java.util.LinkedList$Node`. Since Java applies type erasure [3, 4], no information about generics is available at run time and thus we do not include generics in the DSL. `java.util.LinkedList` represents the *head* of the data structure and has to be marked with one of the head keywords (such as DS). Internal parts of data structures such as `java.util.LinkedList$Node` are not marked with a keyword. Similar to Java syntax, the name of the type is followed by a pair of curly braces. These contain a set of types (separated by semicolons) that may be referenced by the respective data structure part. We call this set of types *pointed-to types*. For example, an instance of `java.util.LinkedList` may point to instances of `java.util.LinkedList$Node` (line 2), which in turn may point to instances of `java.util.LinkedList$Node` instances (line 5), and so on. Line 6 presents two special language features: (1) a star (i.e., `*`) can be used as a wildcard within the name of a pointed-to type and (2) enclosing a type in parentheses declares it as a *leaf*. The term `*` denotes a leaf of any type. Leaf information is used during data structure instance detection to determine the boundaries of a data structure. The DSL also supports namespaces which makes it

possible to omit package declarations in type names. For example, the right side of Figure 5 shows a minimized version of the data structure description. Line 1 defines the namespace `java.util`, thus we can omit the package name for the described types (line 2 and line 5). Since `java.util.LinkedList` only references the list head, we do not have to specify the exact pointed-to type but may use a wildcard instead (line 3). For `java.util.LinkedList$Node`, we may again omit the package name in the pointed-to type (line 6).

3.1.3 Implementation.

To implement our DSL, we used the compiler generator `Coco/R` [21]. It takes an attributed grammar (in EBNF) of a source language and generates a scanner and a recursive descent parser for it. We chose this approach because it allowed us to rapidly prototype first versions of the DSL and to remain flexible in extending the language's grammar with new production rules. The full grammar can be downloaded here¹.

3.2 Parsing Data Structure Descriptions and Detecting Instances

3.2.1 Assigning Data Structure Descriptions to Types.

Before instances of data structures can be detected in a heap state, the data structure descriptions have to be parsed and assigned to their corresponding types. Types without a data structure description are assigned a non-head dummy description that does not declare any pointed-to types. They will always act as leaves in data structures. Array types of reference types are an exception to this rule. They are assigned a non-head dummy description too, but they declare `*` (any type) as their pointed-to type. After this step, every type is equipped with its corresponding data structure description.

3.2.2 Resolving Type Names.

As described in Section 3.1.2, every type's data structure description defines a set of *pointed-to types* that belong to the data structure. These type names may contain wildcards and have to be resolved to all types matching the name pattern. For example, if a type is defined to have a pointed-to type `*Node`, this has to be resolved to the set of all types whose name ends with `Node`. The pointed-to type `*` is not resolved into the set of all types (which would lead to enormous memory overhead), but to `java.lang.Object`.

3.2.3 Detecting Instances.

A reconstructed heap contains information about the objects that are live at a certain point in time (e.g., their types), as well as their references between each other. First, the algorithm filters and remembers all objects that are data structure heads, i.e., objects whose types have a *head* data structure description assigned. Then, to determine which objects belong to a certain data structure, the head's pointers are followed recursively. For every object that is met along the way, the first matching branch of the following four is taken:

- (1) *The object's type is part of its referencing object's set of pointed-to types and is a data structure head type.*

This is the case if a data structure points to the head of another data structure. The head object is treated as a leaf of the referencing data structure and the descent is stopped. A

typical example for this is Java's `HashSet`, which is backed by a `HashMap`. Figure 6 visualizes this pattern. A `HashSet` only consists of two objects: The head of the hash set and the head of the hash map.

- (2) *The object's type is part of its referencing object's set of non-leaf pointed-to types.*

In this case, the object belongs to the referencing data structure, and the descent is continued with its children. For example, this is the case for `LinkedList$Node` instances within a `LinkedList`.

- (3) *The object's type is part of its referencing object's set of leaf pointed-to types.*

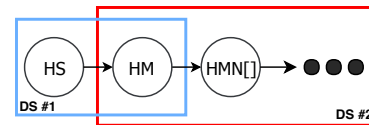
This means that the object belongs to the data structure, but it is a leaf and thus the descent is stopped. For example, the objects that have been added to a `LinkedList`, i.e., those which are referenced from a `LinkedList$Node` instance, are such leaf objects.

- (4) *The object's type is not part of the referencing object's set of pointed-to types.*

This means that the object does not belong to the data structure at all, and thus the descent is stopped.

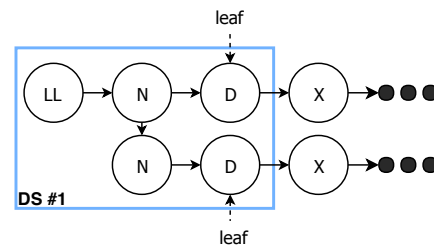
Every visited object is marked to avoid multiple visits.

For example, Figure 7 shows a `LinkedList` that has been detected using the description in Figure 5. The traversal starts at the head `LL`. Next, the first `N` instance is visited. The data structure description of `LinkedList$Node` then tells us to follow further `N` nodes (left side, line 5), or to visit any other object as a leaf without continuing the recursive descent (left side, line 6). Thus, the first `D` instance and the second `N` instance are visited, continuing the descent from the `N` object. As a last step, the second `D` object is visited as a leaf.



HS = HashSet, HM = HashMap, HMN[] = HashMap\$Node[]

Figure 6: A `HashSet` only consists of two objects: The set's head (HS) and the contained hash map's head (HM).



LL = java.util.LinkedList, N = java.util.LinkedList\$Node, D = Data, X = X

Figure 7: A `LinkedList` data structure instances that consists of the head (LL), two nodes (N) and two data objects (D).

¹<http://ssw.jku.at/General/Staff/Weninger/AntTracks/ICPE19/DSL.atg>

Name	Objects		Retained size	
	#	%	#	%
▼ Overall	81,710	100.0	9.6 MB	100.0
ConcurrentHashMap\$Node	242	0.3	4.4 MB	45.5
ConcurrentHashMap	12	0.0	4.4 MB	45.5
ConcurrentHashMap\$Node[]	10	0.0	4.4 MB	45.5
HashMap	22	0.0	4.4 MB	45.4
HashMap\$Node[]	16	0.0	4.4 MB	45.4
HashSet	3	0.0	4.4 MB	45.3
HashMap\$Node	5,038	5.2	4.3 MB	45.0

(a) Object view

Name	Objects		Retained size	
	#	%	#	%
▼ Overall	62	100.0	9.2 MB	100.0
ConcurrentHashMap	12	19.4	4.4 MB	47.7
LinkedList	23	2	3.7 MB	39.9
TreeMap	11	6	1.1 MB	11.8
Properties	3	4.8	25.3 kB	0.3
HashMap	19	30.6	9.8 kB	0.1
Stack	3	4.8	7.4 kB	0.1
ArrayList	6	9.7	456 B	0.0

(b) Data structure view

Figure 8: AntTracks’s object-based heap state view compared to AntTracks’s data-structure-based heap state view.

3.3 Heap State Analysis

Once all data structures have been detected in a certain heap state, the user may utilize this information to investigate potential memory leaks. In AntTracks, a certain heap state can be investigated by applying user- or predefined classifiers on the objects in the heap in order to group them. New classifiers have been developed that utilize the newly gained data structure information for classification. In the following, we present the application of some of these new classifiers, which can be used for top-down analysis as well as for bottom-up analysis. Furthermore, we present the *data structure view*, a new feature to ease top-down analysis.

3.3.1 Top-down Heap Analysis.

In AntTracks, when inspecting a heap state, all objects are initially classified by their types. This is visualized in Figure 8a. Since we are looking for the root cause of a memory leak, i.e., for those objects that keep a lot of other objects alive, we sorted the types by their retained size (i.e., by their ownership). However, the table in Figure 8a still mostly shows internal parts of data structures. These are often inaccessible for developers and are thus of minor interest when looking for the root cause of a memory leak.

Data structure view. We introduce the *data structure view*, a view on the heap state that filters out every object that is not the head of a previously defined data structure. Additionally, data structures that are completely contained in another data structure (i.e., owned by another data structure) are hidden. A typical example for such hidden data structures are HashMaps that are completely contained in a HashSet (as previously shown in Figure 6).

When applying the data structure view on the same heap state as in Figure 8a, only a small fraction of the original heap objects remains visible, as can be seen in Figure 8b. In this example, internal objects such as ConcurrentHashMap\$Node instances are hidden by the data structure view. Additionally, some data structures, such as three HashSets and three HashMaps, are not shown since they are completely contained in another data structure. Depending on the application, data structure head objects often make up far less than 1% of the application. Also the number of object groups, i.e., entries in the classification tree, is greatly reduced. In the example of Figure 8b, it is now easy to tell that among all data structures, instances of ConcurrentHashMap together keep alive 47.7% of the heap. Since the object group of interest still contains 12 objects, we further drill down, i.e., we perform a top-down analysis, as shown in Figure 9. First, we inspect the allocation sites of the maps.

Name	Objects		Retained size	
	#	%	#	%
1 ▼ Overall	62	100.0	9.2 MB	100.0
2 ▼ ConcurrentHashMap	12	19.4	4.4 MB	47.7
3 ▼ DataStructureDevelopmentExample.main...	11	6	4.4 MB	47.5
4 ▼ Data structure leaves	5,003	100.0	4.2 MB	100.0
5 ▼ DeepLongData	5,000	99.9	4.2 MB	100.0
6 DataStructureDevelopmentExample...	5,000	99.9	4.2 MB	100.0
7 Integer	1	0.0	16 B	0.0
8 HashMap\$KeySet	1	0.0	16 B	0.0
9 Object	1	0.0	16 B	0.0
10 ClassLoader.<init>(Void, ClassLoader...	23	2	6.2 kB	0.1

Figure 9: Top-down analysis splits object groups until they are small enough to be analyzed in detail.

Line 3 reveals that the map that has been allocated in method main of class DataStructureDevelopmentExample keeps alive 47.5% of the heap on its own.

Leaf classifier. At this point, the user may want to obtain information about the map’s leaves, i.e., the actual key and data objects contained in the map. The new *leaf classifier* enables users to classify the leaves of a data structure using any classifier combination. For example, in Figure 9, the *leaf classifier* has been used to classify all leaf objects by their types and allocation sites. This clearly identifies the leaves of type DeepLongData (line 5) that have been allocated in the class DataStructureDevelopmentExample (line 6) to consume the most memory. Knowing which data structure keeps a large number of objects alive, as well as which leaf objects within this data structure take up the most memory, should suffice to fix the memory leak.

3.3.2 Bottom-up Heap Analysis.

In top-down analysis, users search for objects or object groups that keep many other objects alive. An alternative approach is the bottom-up approach. Tool users may search for objects that exist in large quantities and then want to find out which other objects keep them alive.

Our approach performs bottom-up analysis at a higher level of abstraction to reduce complexity. Instead of analyzing which *objects* keep the object group of interest alive, we suggest to look for the *data structures* that keep that group alive. For example, Figure 10 shows a classification tree that has been used to perform bottom-up analysis in AntTracks. On the first tree level (line 2 and line 6), objects have been classified by type. This way, we can see that about 50% of heap is kept alive by objects of type DeepLongData (line 2).

Name	Objects		Retained size	
	#	%	#	%
1 Overall	33,628	100.0	8.3 MB	100.0
2 DeepLongData	5,000	14.9	4.2 MB	50.1
3 Data structures	1,100.0		4.4 MB	100.0
4 ConcurrentHashMap	1,100.0		4.4 MB	100.0
5 DataStructureDevelopmentExample.main(...)	1,100.0		4.4 MB	100.0
6 DeepCharData	1,000	3.0	2 MB	24.5

Figure 10: Bottom-up analysis is used to find those objects that keep a given set of objects alive.

Containing data structures classifier. The next step in our approach is to inspect those data structures that contain the objects of interest. To support this, the new *containing data structures classifier* has been developed. This classifier takes an object group and collects the heads of all data structures that contain these objects. These head objects can then be classified using any classifier combination. The information about which objects are contained in the different data structures is obtained and remembered during data structure instance detection, as explained in Section 3.2.3.

For example, the *containing data structures classifier* has been used on the DeepLongData objects in Figure 10 to classify the containing data structure heads by their types, followed by their allocation sites. As we can see, all DeepLongData objects are contained in a single data structure (line 3), which is of type ConcurrentHashMap (line 4) and has been allocated in class DataStructureDevelopmentExample (line 5). This map could now again be investigated further by using the top-down approach described in the previous section.

3.4 Data Structure Growth Analysis

The analysis of data structures at a single point in time may already yield useful insights on the structure of the heap. Growth analysis further supports the user in the search for memory leaks. It considers the growth of data structures over time, which makes it even easier for users to spot those involved in memory leaks. This section describes how objects can be tracked over time using trace-based approaches such as AntTracks. In addition, we present metrics to analyze data structure growth as well as metric patterns based on different growth types. Finally, we present how AntTracks’s existing classification system has been integrated into the new growth analysis. This way, users can, for example, detect data structures that grew over time, tell which set of leaf objects grew the most, and finally where these leaves have been allocated.

3.4.1 Data Structure Instance Tracking.

We argue that trace-based approaches are better suited for temporal analyses than snapshot-based ones. Trace-based approaches can derive temporal information on the *object-level*, while snapshot-based approaches have no concept of *object identity*. Using only snapshots without additional information (e.g., object tagging), it is not possible to decide whether an object that was alive in a certain snapshot still lives in a later snapshot. Figure 11 illustrates this. In a snapshot-based approach, it can only be inferred that both snapshots contained one X object, but not whether these two objects are actually the same instance.

Using AntTracks, we are able to derive this information by replaying the recorded GC move events. Furthermore, we are able

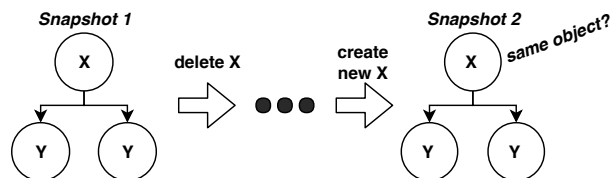


Figure 11: Analysis based on multiple snapshots lacks information on the *object-level*.

to reconstruct the heap object reference graph, i.e., all references between objects, for both points in time. Using this knowledge, we can specifically search for data structures which (1) survived a certain time window (objects that have died cannot be the root cause of a memory leak) and (2) reference / keep alive more objects than before.

The workflow for data structure instance tracking consists in the following steps:

- (1) The user selects two garbage collection points between which the data structure growth analysis should take place.
- (2) The heap is reconstructed for the first point in time and is stored. The addresses of all data structure heads in this heap, i.e., the *start addresses*, are stored as well.
- (3) If a data structure head dies during a garbage collection, we stop tracking it. For surviving heads, their new addresses (which can be reconstructed from GC move events) are stored alongside their start addresses.

Following this algorithm until the end of the selected time window, we obtain (1) the reconstructed heap state at the start, (2) the reconstructed heap state at the end, and (3) a list of all data structures that survived, i.e., their start and final addresses.

3.4.2 Growth Metrics.

For both points in time, i.e. the start and the end of the selected time window, various metrics can be calculated for every data structure head. Subsequently, the absolute growth (both for object count and byte count) can be calculated for each metric. In this section, we present those metrics that have proven most useful in identifying problematic data structures. For simplicity, we refer to both, the object count and the byte count, as *size*.

Retained size growth. The retained size denotes *ownership* and is calculated from the GC closure, as explained in Section 2.2.3. A large retained size means that if the head of the data structure were collected by the garbage collector (i.e., all references to it were removed), a large number of objects / bytes could be collected with it. If the retained size of a data structure grew considerably between two points in time, it is a strong indication for the fact that the data structure is involved in a memory leak. By default, AntTracks’s data structure growth analysis sorts all data structures by this growth metric. This allows us to highlight those data structures that have a suspiciously large growing ownership.

Transitive size growth. A data structure’s *transitive size* denotes *reachability*, i.e., how many objects / bytes (inside or outside the data structure) can be reached from it. At a first glance, the growth of a data structure’s reachability may not seem very useful as a metric on its own. Nevertheless, we will show that it becomes a valuable metrics as soon as it is put into relation with other metrics.

(Deep) Data structure size growth. The two previous metrics ignored data structure boundaries. Now, assuming a list with only a few objects inside it. If the list itself does not grow, i.e., no new objects are added, but its data objects grow, the metrics mentioned above would change.

This is why we introduce the two new metrics: *data structure size* and *deep data structure size*. These metrics are calculated from the newly proposed closures *data structure closure* and *deep data structure closure*. The metrics are supposed to show whether the data structure itself grew, i.e., whether new objects have been added to it.

The *data structure closure* contains all objects that belong to the given data structure. However, objects that belong to other data structures within the given data structure are not included. On the other hand, the *deep data structure closure* also includes objects that are part of such contained data structures. For example, let us revisit the `HashSet` from Figure 6. Every hash set’s data structure closure contains only two objects: The `HashSet` object itself and the contained `HashMap`. A hash set’s deep data structure closure, however, also contains all objects that belong to the hash map, and, if the hash map contains further data structures, also the objects that belong to them.

Heap growth portion (HGP). While it is possible to work purely with absolute growth metrics, our evaluation has shown that metrics are easier to interpret when they are put into relation to the absolute heap growth. Given that the overall heap size increased, the *heap growth portion* (i.e., the portion by which the growth of a specific data structure contributes to the overall heap growth) can be calculated for every metric as shown in Equation 1.

$$HGP_{metric}(ds) = \frac{\Delta metric(ds)}{\Delta heapsize} \cdot 100 \quad (1)$$

For example, $HGP_{retained}(ds)$ puts the retained size growth $\Delta retained(ds)$ of data structure ds into relation with the overall heap growth $\Delta heapsize$.

Assume that the overall heap size increased by 1GB and a list’s retained size increased by 0.7GB. This would result in a $HGP_{retained}$ value of 70%, i.e., the ownership growth of this data structure contributes 70% to the total growth of the heap, which is a strong indication that this list causes a memory leak.

3.4.3 Metric Patterns.

This section discusses five typical metric patterns (see Figure 12) that may occur in various applications. Based on these patterns, users can easily identify the growth type of a data structure, and can determine how to proceed with the investigation.

Single-ownership container growth. If a data structure has a strong retained size growth in combination with a strong (deep) data structure size growth, we can infer two important properties: (1) New objects have been added to the data structure (i.e., container growth), and (2) the data structure keeps the newly added objects alive (i.e., single-ownership). This indicates that it is possibly involved in a memory leak. Nevertheless, this type of memory leak is rather easy to resolve, since the accumulating objects are kept alive by this data structure alone. This means that the leaf objects can be collected by the GC as soon as they are removed from the data structure.

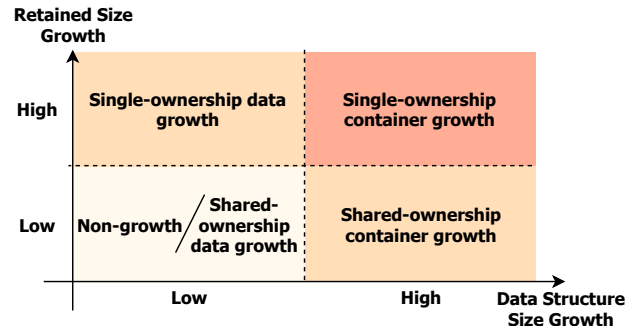


Figure 12: Five common metric patterns.

Inspecting the allocation sites of the data structure as well as of the accumulating leaf objects should yield enough information to identify the source code locations of interest.

Shared-ownership container growth. Similar to *single-ownership container growth*, new objects have been added to the data structure. Yet, the newly added objects are not kept alive by this instance alone. Other data structures may be involved as well.

To find a way to free the newly added objects, the tool user has to analyze the data structure’s leaves in more detail. It is not enough to just remove the leaves from this data structure to make them eligible for garbage collection. They have to be removed from *all* their containing data structures. To find these containing data structures, bottom-up analysis (as shown in Section 3.3.2) can be performed.

Single-ownership data growth. We use this term for a strong retained size growth in combination with a weak (deep) data structure growth. In contrast to *single-ownership container growth*, not the data structure itself is growing (i.e., no or only few new elements have been added). Instead, the ownership over the contained data grew. There are two explanations for this.

One possible reason is that the already owned data grew. For example, imagine a list that stores and owns 100 objects which do not reference any other objects. Over the analyzed time window, the list has not been extended, but the contained objects now reference other objects. This has the effect that the data structure size remains unchanged while the retained size grows.

Another possibility is that multiple data structures share ownership on data objects (or parts of them). The retained size of all involved data structures would be small at the start of the selected time window. However, the retained size would grow for one data structure if the shared ownership changed to single-ownership, e.g., because the shared data has been removed from all data structures except this one.

Non-growth / Shared-ownership data growth. There are two possible patterns for the case that neither the retained size nor the (deep) data structure size grew considerably. They are distinguished based on their deep size growth.

We call the first one *non-growth* data structures. In addition to low retained size growth and low (deep) data structure size growth, also the deep size did *not* change considerably. This is the case when the size of a data structure approximately stayed the same, i.e., neither were many new objects added to the data structure nor

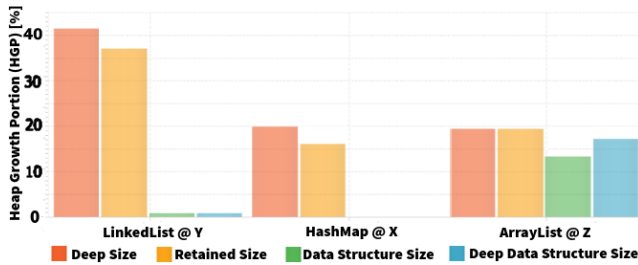


Figure 13: Bar chart to give a quick impression on data structure growth.

did the contained data grow considerably. Thus, data structures classified as *non-growth* do not contribute to a memory leak in the selected time window.

If, however, the deep size *did* grow, the data structure growth can be classified as *shared-ownership data growth*. This means that the data structure itself did not grow, but its data became larger (similar to single-ownership data growth). Yet, in contrast to single-ownership data growth, the data structure does not own the newly referenced objects.

3.4.4 Visualization and Classification.

At this point, all data structures have been tracked over the selected time window and their growth metrics have been calculated. Now these metrics have to be visualized to users in a way that allows them to decide which data structures they should investigate in more detail.

Visualization. AntTracks presents the data structure growth analysis results to the user in two ways. The first one is a bar chart which displays the *HGP* values for deep size growth, retained size growth, data structure growth and deep data structure growth. The bar chart shows the ten data structures with the strongest growth of the currently selected metric. By default, the data structures are sorted and selected based on the retained *HGP*, but users may change this sorting. This visualization gives a quick overview of the metric combinations of those data structures that had the strongest growth of the selected metric. An example is given in Figure 13.

More detailed analysis is possible using a tree table view, similar to the one used in heap state analysis. Every data structure is presented by a single record. The table columns shown by default are absolute and *HGP* values of deep size growth, retained size growth, deep data structure size growth, as well as the object count. By default, the data structures are sorted by their retained size growth, but the user may change the sorting order to any other metric. This way, the data structures can be searched for metric combinations.

Classification. If a suspicious data structure is detected, the user can use AntTracks’s classification system to gain further insight on it. Using the selected classifiers, the data structure is then classified twice: Once at the start of the selected time window, and once at the end. The difference between the two resulting classification trees is then calculated and visualized. This is especially useful to analyze a data structure’s leaf growth behavior. A typical approach is to first classify the data structures by their types, then by allocation sites, followed by the leaf classifier which classifies the leaves by their types and allocation sites. Figure 14 shows an example of this. First,

	Name	Objects		Retained size	
		Before	After	Absolut...	HGP [%]
1	java.util.LinkedList [4,137,788,648]	1	1	960 kB	19.3
2	DataStructureDevelopmentExample.main(...):24	1	1	960 kB	19.3
3	Data structure leaves	5,000	20,000	600 kB	12.1
4	FlatData	5,000	20,000	600 kB	12.1
5	DataStructureDevelopmentExample.main(...):175	0	5,000	200 kB	4.0
6	DataStructureDevelopmentExample.main(...):255	0	5,000	200 kB	4.0
7	DataStructureDevelopmentExample.main(...):303	0	5,000	200 kB	4.0
8	DataStructureDevelopmentExample.main(...):91	5,000	5,000	0 kB	0.0

Figure 14: Tree table view that displays the growth of a data structure using a given classification.

one can see that the LinkedList’s retained *HGP* is about 19%. In line 2, the list’s allocation site is shown. Then, the leaf classifier has been applied (line 3). The *Objects* column shows that the number of leaves raised from 5,000 leaves at the start of the time window to 20,000 leaves at the end of the time window. The *Retained size* column shows that this increase in the number of leaves accounts for about 12% of the overall heap growth. Line 4 shows that all of these leaves are of type `FlatData`. Lines 5 to 8 show the various allocation sites of the leaves. It can be seen that the new leaves have been allocated at three different allocation sites (line 5 to 7). Line 8 conveys the information that the number of leaves that have been allocated at this particular allocation site has not changed.

4 APPLICATION TO CASE STUDIES

To evaluate the usefulness of data structure descriptions and their usage in heap analysis over time, we applied them on two different real-world systems.

The first system is *Dynatrace easyTravel* [9]. Dynatrace focuses on application performance monitoring (APM) and distributes *easyTravel* as their state-of-the-art demo application. It is a multi-tier application for a travel agency, using a Java backend. An automatic load generator can simulate accesses to the service. When *easyTravel* is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

The second system is *AntTracks itself*. AntTracks is under constant development by the authors, as well as by students that do projects with and within AntTracks. Lately, we were dealing with an increasing memory footprint over time during the parsing of trace files. To find the root cause of this memory leak, we analyzed AntTracks using AntTracks’s data structure growth analysis, and document this scenario here.

4.1 easyTravel

EasyTravel was executed on the AntTracks VM, which generated a trace file. This trace file was then opened in the AntTracks Analyzer. After parsing the trace file, multiple charts are presented to the user, displaying the application’s memory behavior. For example, Figure 15 shows the number of allocated objects separated by heap space type (y-axis) over time (x-axis). In most generational garbage collectors, objects are allocated in a heap space called *eden*, are then moved to a *survivor* space if they survive at least on garbage collection, and are eventually promoted to an *old* space after a certain number of garbage collections. The growing number of objects in the old space (red) clearly indicates that the heap consumption grew over time.

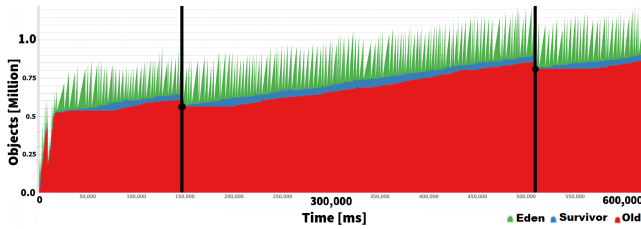


Figure 15: Object count evolution that hints at a memory leak in easyTravel.

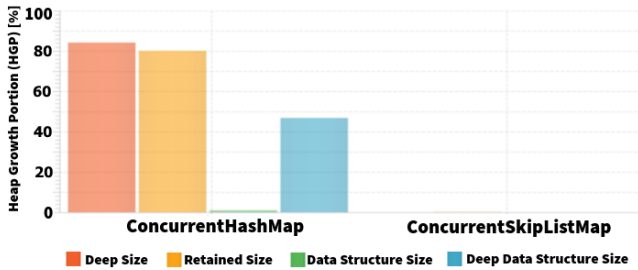


Figure 16: Metric growth bar chart that highlights the ConcurrentHashMap as memory leak suspect due to its strong retained size growth.

Name	Objects	
	Before	After
Overall	25,748	25,748
java.util.concurrent.ConcurrentHashMap...	1	1
JourneyService.findLocations(String, ...)	1	1
Data structure Leaves	32,058	136,224
Location	31,781	135,026
GeneratedConstructorAccessor33.n...	31,769	135,014
Constructor.newInstance(Object[]...	12	12
JourneyService\$QueryKey	276	1,196
JourneyService.findLocations(Str...	276	1,196
ConcurrentHashMap\$CounterCell	0	1
Collections\$EmptyList	1	1

Figure 17: Classification of the conspicuous map based on allocation site followed by leaf classification based on leaf types and leaf allocation sites.

To perform data structure growth analysis, the user can specify a time window by selecting two points in time (the vertical black lines in Figure 15). In this example, we selected the end of two major garbage collections. At these points we can assure that all heap objects that were not reachable anymore have been collected by the GC.

Once all surviving data structures have been tracked over the selected time window, their growth are visualized in the bar chart (Figure 16) and the tree table view. What can be seen at a glance is that the retained HGP of the ConcurrentHashMap is about 80%. This clearly identifies this specific ConcurrentHashMap as the memory leak culprit. What can further be derived from the other metrics is that nearly no new objects have been added to the data structure itself (i.e., very low data structure size growth). Yet, since the deep data structure size grew strongly, we can derive that the map must contain further data structures, and that these data structures have grown. As explained in Section 3.4.3, this can be categorized as *single-ownership container growth*.

Since we now know that this concurrent hash map is the major suspect for the memory leak, we want to gain more information about it until we are able to fix the leak. Figure 17 shows the classification tree that we used to analyze the map. The first classifier that has been used on the map is the *allocation site classifier*. It tells us that the map has been allocated in the method `findLocations` of class `JourneyService` (line 3). The second classifier that has been applied is the *leaf classifier*. As the name suggests, this classifier can be used to inspect the leaves of a data structure. The classifier has two modes: either *own leaves* or *deep leaves*. In the first mode, the classifier would inspect only the leaves of the map itself, without checking the leaves of contained data structures. Since we know that the map contains further data structures, the *deep leaves* mode has been selected. Furthermore, the classifier was configured to classify all leaf objects based on their types and allocation sites. Looking at the *Objects* column, we can see that the number of leaves grew from about 32,000 to about 136,000 (line 4). We can further see that nearly all of these leaves are of type `Location` (line 5). Unfortunately, their allocation sites are not useful, since they are somewhere hidden in a framework (line 6). The second growing leaf type is `JourneyService$QueryKey` (line 7). These leaves have been allocated in method `findLocations` of class `JourneyService`.

Even though we had no prior knowledge about the system, we decided at this point that we gained enough insight to investigate the memory leak on the source code level. To prevent the proliferation of the concurrent hash map, we must prevent that its `Location` and `JourneyService$QueryKey` leaves accumulate. In the source code, the map of type `ConcurrentHashMap<QueryKey, Collection<? extends Location>>` was easily found. In the method `findLocations` (the allocation site of the accumulating `QueryKey` instances) we found that the map should have served as a cache for location searches. Once a search was executed for a given `QueryKey`, the key was stored in the map, alongside its search result (a `Collection<Location>`). Subsequent searches for the same key should have found the respective entry in the map. Yet, `QueryKey` neither implements `hashCode` nor `equals`. Thus, every request resulted in a cache miss and consequently a new cache entry, which led to this typical memory leak.

4.2 AntTracks

Figure 18 shows AntTracks's object count evolution during trace file parsing. Similar to easyTravel, we can see an increase of objects in the old generation of the heap.

Yet, in contrast to easyTravel, which had to be analyzed without prior knowledge about its data structures, this time we could use the data structure DSL to describe AntTracks's most important data structures prior to analysis. Since the leak occurred during trace parsing, the first data structure that was described was AntTracks's internal representation of heap states. To mimic the structure of the real heap, AntTracks separates the heap under reconstruction in a number of `Space` instances, which are further divided into `LAB` (local allocation buffer) instances, which then store the actual information about objects, mostly using arrays. We also described AntTracks's data structure that keeps track of symbols information such as type names, allocation sites, and so on, since this data could also have been corrupted during trace parsing.

```

1 namespace java.util { // By default shipped with AntTracks
2   HashMap$Node {
3     HashMap$Node;
4     (*);
5   }
6   DS HashMap {
7     HashMap$Node[];
8   }
9   // ... other java.util classes
10 }
11
12 namespace at.jku.anttracks.util { // Added for specific use case
13   DS ApplicationStatistics {
14     java.util.HashMap; // HashMap<Thread, MeasurementGroup> and others
15   }
16   ApplicationStatistics$MeasurementGroup {
17     *; // Various internal objects, including List<Measurement>
18   }
19   ApplicationStatistics$Measurement { }
20 }

```

Listing 1: Description of AntTracks’s data structure to track the execution time of certain code segments.

AntTracks has an internal performance evaluation feature called `ApplicationStatistics`, which is implemented as a singleton. It supports to creation of `Measurement` objects, which can be used to evaluate how much time is spent by which thread in certain code segments. Multiple measurements are then grouped together in a `MeasurementGroup` instance. These data structure parts have also been described, and their descriptions can be seen in Listing 1.

After calculating the data structure growth over the time window selected in Figure 18, an overview bar chart (Figure 19) and a tree table view is shown. By looking at the bar chart, it becomes clear that the memory leak is caused by the `ApplicationStatistics` instance. The metric pattern is akin to that of the memory leak found in `easyTravel`: a typical *single-ownership container growth*. It may be noteworthy to mention that a data structure’s *HGP* values can be above 100%, as can be seen in Figure 19, where the `ApplicationStatistics`’s retained *HGP* is around 115%. In this case, the overall heap grew by about 100MB, while the `ApplicationStatistics`’s ownership grew by 115MB, which can happen if previous multi-object ownership changed to single-object ownership, as explained in Section 3.4.3.

Figure 20 shows the classification tree used to analyze the memory leak. Since `ApplicationStatistics` is implemented as a singleton, it is not necessary to check its allocation site. The result of the *leaf classifier* already provided enough information to resolve the memory leak. The classifier has been configured to classify each leaf by its type, followed by its allocation sites, as well as the call sites of the allocating methods. Line 3 shows that the overall number of leaves has skyrocketed from about 6.6 million to 10.2 million. Nearly all of the leaves are of type `Measurement` (line 4), and all of them have been allocated in the `ApplicationStatistic` class (line 5). To further distinguish the measurements, the methods that called the allocating method can be inspected. This reveals that two call sites, both located in the method `parseGCRootPtr` of class `TraceParserSlave` (line 6 and 7), caused the extensive `Measurement` allocations.

Checking the `TraceParserSlave` class, the instrumented parts within `parseGCRootPtr` were easily detected. These parts were

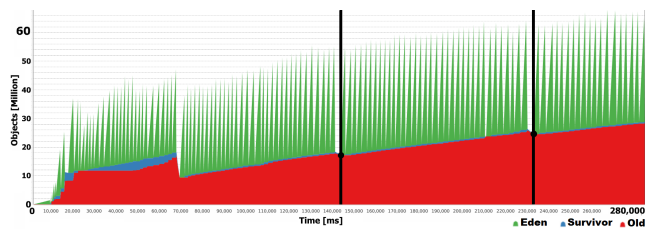


Figure 18: AntTracks’s object count evolution shows a similar pattern as in `easyTravel`.

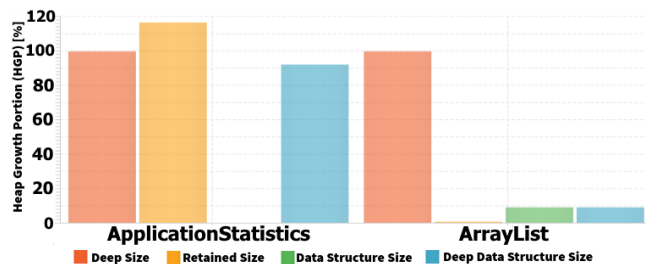


Figure 19: Bar chart clearly showing the single-ownership container growth of the `ApplicationStatistics` instance.

Name	Objects	
	Before	After
Overall	11,486	11,486
at.jku.mevss.anttracks.util.ApplicationStatistics...	1	1
Data structure leaves	6,632,638	10,263,731
ApplicationStatistics\$Measurement	6,632,610	10,263,703
ApplicationStatistics.createMeasurement(Strin...	6,632,610	10,263,703
TraceSlaveParser.parseGCRootPtr(EventType,...	4,187,985	6,454,359
TraceSlaveParser.parseGCRootPtr(EventType,...	2,340,602	3,662,890
TraceSlaveParser.parseGCRootPtr(EventType,...	8	14
MutableLong	16	16
Thread	12	12

Figure 20: Classifying the leaves of `ApplicationStatistic` by type, allocation site, and call site.

frequently called and thus created a vast amount of `Measurement` instances. Since they were not essential to AntTracks’s functionality, they were simply removed to resolve the memory leak.

5 RELATED WORK AND STATE-OF-THE-ART

To support memory leak detection as well as to facilitate memory leak resolving, various approaches and tools have been developed over the last years. Šor and Srirama [43] classify these approaches into the following groups:

- (1) *Online approaches* that actively monitor and interact with the running virtual machine, separated into approaches that
 - (a) *measure staleness* [2, 11, 23, 24, 41]. Staleness is not a quantifiable metric, instead, the longer an object is not used, the more stale it becomes. The idea behind approaches that measure staleness is that objects that do not get collected by the GC for a long time but become stale are more likely to be leaking than non-stale objects. The challenge that these approaches face is that object access tracking is extremely expensive.
 - (b) *detect growth* [5, 12, 13, 30, 31]. These approaches group the live heap objects (mostly based either on their types or allocation sites) and detect growth using various metrics.

These metrics range from simple absolute count differences between allocations and deallocations [5] to more complex ones based on the structure of the object reference graph [12, 13].

- (2) *Offline approaches* that collect information about an application for later analysis, separated into approaches that
 - (a) *analyze heap dumps as well as other kinds of captured state* [15, 18–20]. Compared to online approaches, offline approaches often perform more complicated analyses based on the object reference graph, involving graph reduction, graph mining and ownership analysis.
 - (b) *use visualization* to aid manual leak detection [6, 22, 25].
 - (c) *employ static source code analysis* [7, 42].
- (3) *Hybrid approaches* that combine online features as well as offline features [8, 26, 40].

For example, one of the approaches most similar to our approach is *container profiling* by Xu and Rountev [41]. They also focus on data structures, but instead of detecting growth, they track operations on containers and detect container staleness. Their approach requires ahead-of-time modeling of containers, i.e., the user has to introduce a “glue layer” in the monitored application’s source code that maps methods of each container type to primitive operations (e.g., ADD, GET, and REMOVE). Compared to that, our data structure description mechanism using a DSL is much less invasive.

Future work (see Section 6) encompasses plans to automatically infer data structure descriptions from source code and memory traces. For example, Mitchell and Sevitsky [19] developed *LeakBot*, a tool that performs memory analysis using object aggregation. They present various metrics to detect possible top-level *leak roots*, which may correspond to data structure heads. Jump and McKinley [14] introduced *dynamic shape analysis*, which seeks to *characterize data structures* by summarizing the object pointer relationships into *degree metrics*. Metrics like these may facilitate the process of automatically inferring data structure descriptions.

6 FUTURE WORK

Our data structure description DSL can be used to describe arbitrary data structures. However, users might find it tedious to describe a greater number of custom data structures which they use in their project. To relieve the user of this task, reasonable data structure definitions should be inferred automatically from static information, such as the source code, in combination with dynamic information obtained during trace parsing. The DSL presented in this work would not become obsolete by such a feature because automatically detected data structure descriptions would most likely require corrections or extensions by the user. Moreover, in some cases, users might want to describe undetectable reference patterns as data structures.

In AntTracks, the heap and subsets thereof are currently represented in the form of charts and tree table views. In the future, users should also be able to browse through objects and their reference patterns in a visualized reference graph. However, considering the number of objects and references in heaps of modern applications, visualizing complete reference graphs is infeasible in terms of performance. Moreover, graphs of such dimensions are also impossible to comprehend for users and consequently are not of much use to

locate the root cause of a memory leak. The newly gained information about data structures in the monitored application could be used to greatly reduce the number of nodes and edges that have to be visualized. Instead of displaying every object as a separate node, objects that belong to a given data structure could be collapsed into a single node, ultimately reducing the object reference graph to a *data structure reference graph*. Guided by the metrics that were presented in this paper, users could locate the root cause of a memory leak by visually browsing through such a graph, investigating the data structures they are interested in.

7 THREATS TO VALIDITY

In this paper, we presented five metric patterns that commonly occur during data structure growth analysis. Each of these patterns suggests different analysis steps. This poses two potential threats to validity: (1) Users may find it hard to comprehend all metrics and their patterns without prior training, and (2) one could argue that the presented list of patterns is not exhaustive and that further patterns could be defined (for example, patterns involving shrinking metrics have not been discussed in this paper). To deal with these threats, future work includes automatic decision making by the tool. New heuristics should be defined that allow the tool to automatically detect metric patterns. Based on detected patterns, the tool could either perform certain classifications or analysis steps automatically, or it could provide suggestions to the user on how to proceed in the analysis, similar to a learning-by-doing tool approach.

Most probably, the major threat to validity of our work is its currently restricted evaluation based on a limited set of use cases. We plan to search for open-source projects that suffered from memory leaks in the past with the goal of building a reference set of real-world applications that could be used to evaluate memory leak detection tools. Using this set of applications, alongside other applications with seeded memory defects, we plan to conduct a user study with our industry partner as well as with university students. In addition to comparing AntTracks to existing tools, e.g., in terms of found memory leaks, we want to gain insight in how well the study participants are able to understand and use existing memory leak detection features, as well as what other features users expect from a memory monitoring tool. This could help the community to improve the quality of memory monitoring tools in general.

8 CONCLUSION

In this paper, we presented a memory leak detection approach that puts data structures into the focus of its analysis. To prove its applicability, we integrated this approach into AntTracks, a trace-based memory monitoring tool.

Our approach encompasses an easy-to-use domain specific language to describe arbitrary data structures, as well as an algorithm that detects instances of those data structures in reconstructed heaps. Further, we presented a new feature in AntTracks called *data structure view*. It hides objects of lower interest, i.e., data-structure-internal objects, during heap state analysis and emphasizes data structure head objects. This reduces the complexity of heap state analysis users have to deal with in state-of-the-art memory monitoring tools. To inspect conspicuous data structures, we developed

new data-structure-specific analysis features to support top-down as well as bottom-up memory analysis.

Our main contribution is a new technique for analyzing the growth of data structures over time: We (1) showed how to use memory traces to track data structures throughout an application's lifetime, (2) introduced metrics that describe various aspects of data structure growth, (3) discussed how certain metric patterns hint at certain types of memory leaks, and (4) presented techniques to prioritize, visualize and analyze data structures that may be the root cause of a memory leak. Data structure growth analysis aims to further ease the analysis of memory leaks by reducing the number of steps a user has to take to identify the root cause of such leaks. Finally, we evaluated the applicability of our approach using two case studies.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

- [1] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*.
- [2] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: Bit-encoding Online Memory Leak Detection. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.
- [3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. 1998. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proc. of the 13th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*.
- [4] Robert Cartwright and Guy L. Steele, Jr. 1998. Compatible Genericity with Runtime Types for the Java Programming Language. In *Proc. of the 13th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*.
- [5] K. Chen and J. Chen. 2007. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Proc. of the 31st Annual Int'l Computer Software and Applications Conf. (COMPSAC '07)*.
- [6] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '99)*.
- [7] Dino Distefano and Ivana Filipović. 2010. Memory Leaks Detection in Java by Bi-abductive Inference. In *Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering (FASE 2010)*.
- [8] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended Analysis for Performance Understanding of Framework-based Applications. In *Proc. of the 2007 Int'l Symposium on Software Testing and Analysis (ISSTA '07)*.
- [9] Dynatrace. 2019. Demo Applications: easyTravel. <https://community.dynatrace.com/community/display/DL/Demo+Applications++easyTravel>
- [10] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proc. of the 40th Int'l Conf. on Software Engineering: Companion Proceedings (ICSE '18)*.
- [11] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*.
- [12] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '07)*.
- [13] Maria Jump and Kathryn S. McKinley. 2009. Detecting Memory Leaks in Managed Languages with Cork. *Software: Practice and Experience* 40, 1 (2009).
- [14] Maria Jump and Kathryn S. McKinley. 2009. Dynamic Shape Analysis via Degree Metrics. In *Proc. of the Int'l Symposium on Memory Management (ISMM '09)*.
- [15] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD '10)*.
- [16] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.
- [17] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE '15)*.
- [18] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *Proc. of the 20th European Conf. on Object-Oriented Programming (ECOOP '06)*.
- [19] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '03)*.
- [20] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications (OOPSLA '07)*.
- [21] Hanspeter Mössenböck, Markus Löberbauer, and Albrecht Wöß. 2019. The Compiler Generator Coco/R. <http://www.ssw.uni-linz.ac.at/Coco/>
- [22] Wim De Pauw and Gary Sevitsky. 2000. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience* 12, 14 (2000).
- [23] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '07)*.
- [24] Derek Rayside, Lucy Mendel, and Daniel Jackson. 2006. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proc. of the Int'l Workshop on Dynamic Systems Analysis (WODA '06)*.
- [25] S. P. Reiss. 2009. Visualizing The Java Heap to Detect Memory Problems. In *5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '09)*.
- [26] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. 2000. Automatic Removal of Array Memory Leaks in Java. In *Proc. of the 9th Int'l Conference on Compiler Construction (CC '00)*.
- [27] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proc. of the 2nd Int'l Workshop on Software and Performance (WOSP '00)*.
- [28] Connie U. Smith and Lloyd G. Williams. 2002. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*
- [29] Connie U. Smith and Lloyd G. Williams. 2003. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*
- [30] V. Sor, P. Oü, T. Treier, and S. N. Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM '13)*.
- [31] Vladimir Šor, Nikita Salmikov-Tarnovski, and Satish Narayana Srirama. 2011. Automated Statistical Approach for Memory Leak Detection: Case Studies. In *On the Move to Meaningful Internet Systems (OTM 2011)*.
- [32] Eclipse Foundation. 2019. Eclipse Memory Analyzer (MAT). <https://www.eclipse.org/mat/>
- [33] Oracle. 2019. The HotSpot Group. <http://openjdk.java.net/groups/hotspot/>
- [34] Oracle. 2019. VisualVM: All-in-One Java Troubleshooting Tool. <https://visualvm.github.io/>
- [35] Peter Wegner and Edwin D. Reilly. 2003. Data Structures. In *Encyclopedia of Computer Science*.
- [36] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.
- [37] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l Conf. on Managed Languages & Runtimes (ManLang '18)*.
- [38] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.
- [39] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 9th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '18)*.
- [40] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '11)*.
- [41] Guoqing Xu and Atanas Rountev. 2008. Precise Memory Leak Detection for Java Software Using Container Profiling. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE '08)*.
- [42] Daocng Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *Proc. of the Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '14)*.
- [43] Vladimir Šor and Satish Narayana Srirama. 2014. Memory Leak Detection in Java: Taxonomy and Classification of Approaches. *Journal of Systems and Software* 96 (2014).

Chapter 5

Visualization

5.1 Drill-down Trend Visualization

This section includes the paper [325] on how to visualize the evolution of memory trees over time in a time-series chart with drill-down functionality.

Paper:

Markus Weninger, Lukas Makor, Elias Gander, Hanspeter Mössenböck:
AntTracks TrendViz: Configurable Heap Memory Visualization Over Time.
In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE 2019, Mumbai, India, April 07-11, 2019.

AntTracks TrendViz: Configurable Heap Memory Visualization Over Time

Work-In-Progress Paper

Markus Weninger

Institute for System Software, CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria

Elias Gander

CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria

Lukas Makor

Johannes Kepler University
Linz, Austria

Hanspeter Mössenböck

Institute for System Software
Johannes Kepler University
Linz, Austria

ABSTRACT

The complexity of modern applications makes it hard to fix memory leaks and other heap-related problems without tool support. Yet, most state-of-the-art tools share problems that still need to be tackled: (1) They group heap objects only based on their types, ignoring other properties such as allocation sites or data structure compositions. (2) Analyses strongly focus on a single point in time and do not show heap evolution over time. (3) Results are displayed in tables, even though more advanced visualization techniques may ease and improve the analysis.

In this paper, we present a novel visualization approach that addresses these shortcomings. Heap objects can be arbitrarily classified, enabling users to group objects based on their needs. Instead of inspecting the size of those object groups at a single point in time, our approach tracks the growth of each object group over time. This growth is then visualized using time-series charts, making it easy to identify suspicious object groups. A drill-down feature enables users to investigate these object groups in more detail.

Our approach has been integrated into AntTracks, a trace-based memory monitoring tool, to demonstrate its feasibility.

KEYWORDS

Memory Monitoring, Heap Growth Analysis over Time, Visualization, Memory Leak Detection

ACM Reference Format:

Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion)*, April 7–11, 2019, Mumbai, India. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3302541.3313100>

ICPE '19 Companion, April 7–11, 2019, Mumbai, India

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Tenth ACM/SPEC International Conference on Performance Engineering Companion (ICPE '19 Companion)*, April 7–11, 2019, Mumbai, India, <https://doi.org/10.1145/3302541.3313100>.

1 INTRODUCTION

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). Nevertheless, memory problems can still occur even in garbage-collected languages. One of the major types of memory problems are memory leaks [4], i.e., objects may remain reachable from GC roots even though they are no longer needed. For example, if a developer forgets to remove objects from their containing data structures, these objects cannot be reclaimed by the GC and will accumulate over time.

State-of-the-art tools, such as VisualVM [14] or Eclipse Memory Analyzer (MAT) [12], perform memory analysis based on a heap snapshot, i.e., a heap dump. They group the live heap objects by their types and display the number of objects and the number of bytes per type in a table, i.e., in a *type histogram*. In addition to that, they support comparing two heap snapshots, showing the increase or decrease of live objects / live bytes per type in a table.

While this information may be sufficient to detect basic memory problems, such analysis approaches have also various shortcomings. First, fixing a memory leak might require more information about the objects besides their types, for example, their allocation sites or the data structures in which they are contained. Second, comparing two snapshots does not reveal general trends in an application's memory behavior. An increase in instances of a certain type between two given points in time does not necessarily indicate a continuous memory growth. To detect trends, the heap has to be compared at multiple points in time, a feature that is not supported by the two tools mentioned. Finally, tools should make it as easy as possible to extract the needed information. Great potential to make data more accessible lies in the use of *data visualization* [5].

In the following, we present a work-in-progress approach on how to visualize continuous memory consumption trends over time. Users can group the heap objects by arbitrary criteria such as their types or their allocation sites and visually inspect the heap evolution per object group. Trends within certain object groups can hint at memory leaks and other memory anomalies. We integrated our approach into AntTracks, a trace-based memory monitoring tool based on the Hotspot Java VM, initially developed by Lengauer et al. [9] and extended by Weninger et al. [18–21].

Our contribution encompasses:

- a technique to derive growth information of heap object groups from memory traces.
- a highly configurable visualization approach for trend analysis. It displays the growth of object groups over time (based on various size metrics) using time-series charts.
- a drill-down feature to re-apply the same visualization approach on a specific subgroup of suspicious objects.
- a working implementation in AntTracks.

2 BACKGROUND

AntTracks consists of two parts: The AntTracks VM, a virtual machine based on the Java Hotspot VM [13], and the AntTracks Analyzer, a memory analysis tool. Since the concepts presented in this paper have been integrated into AntTracks, it is essential to understand how AntTracks works.

Trace Recording and Reconstruction. The AntTracks VM writes information about memory events such as object allocations and object movements executed by the GC into a trace file. It keeps the event size to a minimum and avoids the recording of redundant data [8, 9]. Later, the AntTracks Analyzer can incrementally process such a trace file. It is able to display the overall memory development over time and enables users to reconstruct and inspect the heap state at every garbage collection point [2]. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by.

Heap Object Classification. The AntTracks Analyzer uses *object classifiers* in combination with *multi-level grouping* [20, 21] to enable user-driven heap analysis. An object classifier groups heap objects into multiple object groups according to certain criteria such as their types, their allocation sites, or their allocating threads. For example, the *Type* classifier groups objects by their types, e.g., `java.util.LinkedList`. In multi-level grouping, objects are grouped according to the classification results of multiple classifiers. This results in a hierarchical classification tree. In general, every node in such a tree represents an object group and the amount of objects / bytes classified in the respective sub-tree.

For example, the classification tree in Figure 1 has been created by applying the *Type* classifier, followed by the *Allocation Site* classifier. Overall, the classification tree represents 120,000 objects, 5,000 of them are of type `Object[]`, and 1,000 of these arrays have been allocated at `Stack:init()`.

3 APPROACH

This section covers the basic concepts of our new visualization approach. We show how to reconstruct object group growth information from memory traces, together with a highly configurable time-series-based visualization, and how this visualization can be used to drill-down into specific object groups to gain further insights on their growth behavior.

3.1 Gathering Object Group Information

When investigating an application with memory problems, certain parts of its execution trace will stand out. For example, if the

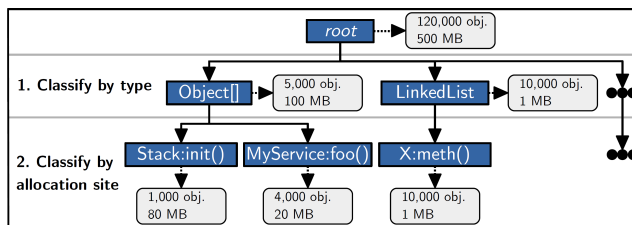


Figure 1: A classification tree that first groups all objects by their types and then by their allocation sites.

memory consumption grew extraordinarily strong between two points in time and does not shrink again afterwards, it indicates that objects have been allocated within this timespan that cannot be reclaimed by the garbage collector later, indicating some type of memory leak. In our approach, users can select such a time window for subsequent analysis.

After selecting the time window of interest, the user has to decide according to which criteria the heap objects should be grouped into object groups. The growth of each object group will later be visualized to aid users in detecting object group growth trends.

Subsequently, the memory trace is incrementally parsed to reconstruct the heap objects within the selected time window. During parsing, the live objects are classified at every garbage collection point using the selected list of classifiers. The resulting classification tree is then stored alongside a timestamp that identifies the respective garbage collection. Every time a new entry is added to this timeline of classification trees, a new time-series data set is generated that is used to update the heap object group memory growth visualization.

3.1.1 Improvements. Instead of classifying the live objects at every garbage collection point within the selected time window, only a subset of the garbage collections can be used for classification to improve performance. The user can select to only classify at every n -th garbage collection, or after at least x seconds have passed since the last classification in the traced application. The benefit is that the classification, which is the most performance-intensive task, can be performed less often. In most cases, existing memory trends will still be apparent.

Future work includes the automatic selection of interesting time windows, thus freeing the user from this task.

3.2 Data Set Generation

To enable growth detection through visualization, the sequence of classification trees first has to be converted into a visualizable data set. The individual classification trees represent the application’s heap state at different points in time. Thus, a time-series-based visualization is the most natural choice. Since the time-series plot is the most frequently used form of graphic design [15], it is well-known and easy to understand. In general, time-series data takes the following form: $D = \{(t_1, y_1), (t_2, y_2), \dots, (t_n, y_n)\}$ [17], i.e., it consists of data pairs where a given point in time t_i has a certain value y_i assigned to it.

To achieve this format, the nodes on the first level of every classification tree, i.e., the object groups formed by the first classifier, are extracted. Next, a time series is created for every distinct node key. For example, if the objects were first classified and grouped

using the *Type* classifier, every type would become a series in the data set. Each series contains one entry (t, y) per classification tree. t is the timestamp assigned to the respective classification tree, and y is the object group’s size, which can be extracted from its tree node within the classification tree.

There are multiple size metrics [18] that users can choose from, either in number of objects or number of bytes:

- *Shallow size*: The number of objects / bytes of an object group, without taking into account any referenced objects.
- *Deep size*: The number of objects / bytes of an object group, including all objects *reachable* from them.
- *Retained size*: The number of objects / bytes of an object group, including all *owned* objects. In other words, it includes all objects that could be freed by the garbage collector if the given object group would be freed.

3.3 Visualization

Depending on the used classifier, the data set can end up containing a large number of series. For example, if the objects have been classified by their types, a series is created for every type, which can easily be several thousands. Yet, most of these series are not of interest when searching for possible memory problems. Thus, our approach supports various techniques to select those series that are of most interest to the user.

To decide which series should be shown, the series are sorted according to a given strategy, and only the top N series are selected. The following sorting strategies are currently supported:

- *Start and End* sorting: The series are sorted by their values at the start or the end of the time window, respectively. This way, the memory evolution of those series, i.e., object groups, that take up the most heap space at the start / end of the time window can be inspected.
- *Average* sorting: The series are sorted by their average y -value. This setting can be used to inspect the growth behavior of those object groups that took up the most heap space throughout the selected time window.
- *Absolute growth* and *Relative growth* sorting: The series are sorted by their absolute or relative increase between the start and the end of the time window, respectively. This enables users to inspect the growth behavior of the object groups that grew the most over the selected time window.

The user can also select if an *Other* series should be shown that combines all object groups that are not visualized as separate series. By default, the *Other* series is shown and the *Absolute growth* sorting with an N -value of 5 is selected. An example of the visualization is shown in Section 4.

3.4 Drill-down

As explained in Section 3.2, the initial visualization extracts the first level of the classification trees and visualizes their object groups’ growth behavior over time. If multiple classifiers have been applied to build the classification trees, a suspicious object group (e.g., a group with a strong growth within the selected time window) can be selected for *drill-down* in the visualization. The drill-down feature re-applies the same visualization technique to the children of the selected object group and displays it in a new chart.

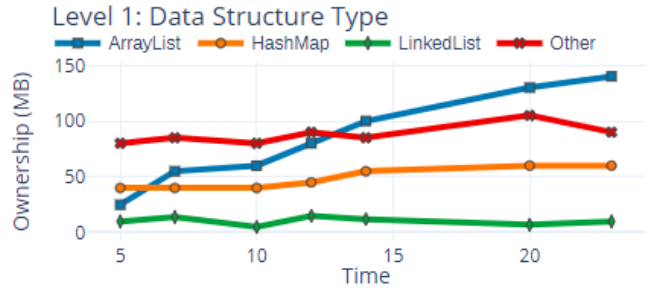


Figure 2: Visualizing the *retained size* of data structure types over time highlights ArrayList as suspect.

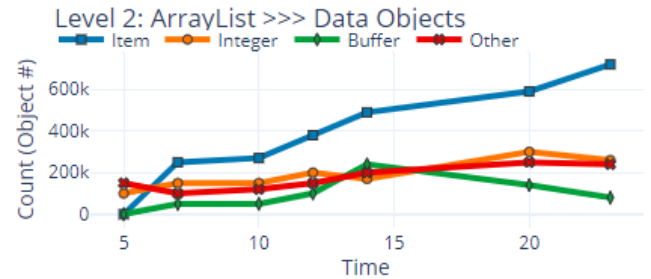


Figure 3: Drill-down showing the development of the *object count* of ArrayList’s data objects.

For example, assume that the *Type* classifier has been used as the first classifier, followed by the *Allocation Site* classifier, creating classification trees similar to the one shown in Figure 1. If the user detects suspicious growth for objects of type `Object[]`, this group can be selected for drill-down. A new data set will be created, based on the various allocation sites at which objects of type `Object[]` have been created, according to the steps explained in Section 3.2. This allocation site data set will then be visualized below the existing chart, using the steps described in Section 3.3.

The various settings, such as the sorting strategy, can be adjusted individually per chart. To make browsing the charts more convenient, interaction features such as zooming are synchronized over all charts.

4 EXAMPLE

In this example, we show a typical way of how to use our visualization approach by demonstrating how to identify and inspect data structure types with growing ownership in AntTracks.

First, we select a time window over which the application’s memory grew considerably. In this example, the basic idea is to (1) first inspect the *retained size growth*, i.e., ownership growth, of data structure types (such as HashMaps), (2) then selecting the type with the highest growth for drill-down, (3) followed by a visualization of the data object growth within this data structures to find out which data objects accumulate the most.

Our example uses one filter and two classifiers to group the heap objects for visualization. We are using the *Data Structure* filter, which only includes data structure head objects (for example lists, maps, etc.) during classification, ignoring other objects. These data structure head objects are then classified by their types. Visualizing the retained size growth, i.e., the ownership growth, of these types results in a chart similar to the one in Figure 2, which shows that

objects of type `ArrayList` have the strongest retained size growth. Thus, this type is selected for drill-down.

As a second classifier we are using the *Data Object* classifier which enables us to analyze the data objects stored in a data structure. Due to paper length restrictions, we group those data objects only by their types. Typically, they would also be grouped by their allocation sites. In Figure 3, the drill-down on the `ArrayList` object group was configured to show the growth of the number of data objects in `ArrayLists` per data object type.

We are now able to easily pinpoint `ArrayList` data structures that contain `Item` objects as the major suspects for a possible memory leak. This information could now be used to investigate the memory problem on the source code level.

5 RELATED WORK AND FUTURE WORK

State-of-the-art tools include, among others, VisualVM [14] and Eclipse Memory Analyzer (MAT) [12], which have been discussed in Section 1.

In their work on the taxonomy and classification of memory analysis approaches in Java, Šor and Srirama [23] highlight the visualization approaches by De Pauw and Sevitsky [3, 10] and by Reiss [11]. The former extracts reference patterns (repetitive reference sequences in a heap object graph) and visualizes them. In addition to that, such reference patterns can also be extracted for those objects that are created between two heap snapshots (e.g., potentially leaking objects), which can then be visually explored. The latter work visualizes the object ownership in a tree-like visualization using shapes, coloring, hatching, hue and saturation. Another approach that compares heap snapshots has been developed by Jump and McKinley [6, 7]. Their tool, Cork, compares the heap object graph structure of two heap snapshots to detect the growth of certain reference patterns between classes.

Future work includes a more thorough evaluation and presentation of our new visualization approach based on real-world scenarios. In addition to that, the approach can still be extended by numerous features. For example, the analysis time windows could be chosen automatically by the tool. Besides the current visualization using line charts, other visualization techniques could also be evaluated based on the same underlying data, such as *small multiples* [16] or as *software cities* [22]. Other typical visualization techniques that can still be further explored involve the representation of aggregated heap objects as graphs [1].

6 CONCLUSION

In this work, we presented a new approach to visualize the growth of heap object groups over time. Trends detected in this visualization can hint at memory problems such as memory leaks involving certain object groups. To construct the underlying data for the visualization, the live heap objects are split into groups based on user-selected criteria (e.g., by their types) at multiple points in time. The evolution of each group over time is then visualized in a time-series chart. The visualization is highly user-configurable based on the user's needs, allowing users to select features such as series sorting, series selection or size metrics. A drill-down feature enables users to select an object group of interest, e.g., a strongly growing group, to classify the objects within this group by another

criterion, and to re-apply the same visualization technique to this group.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

- [1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the 5th Int'l Symp. on Software Visualization (SOFTVIS '10)*.
- [2] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*.
- [3] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proc. of the European Conf. on Object-Oriented Programming (ECOOP '99)*.
- [4] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proc. of the 40th Int'l Conf. on Software Engineering: Comp. Proc. (ICSE '18)*.
- [5] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. 2010. A Tour Through the Visualization Zoo. *Commun. ACM* 53, 6 (June 2010).
- [6] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '07)*.
- [7] Maria Jump and Kathryn S. McKinley. 2009. Detecting Memory Leaks in Managed Languages with Cork. *Software: Practice and Experience* 40, 1 (2009).
- [8] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Wening, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.
- [9] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE '15)*.
- [10] Wim De Pauw and Gary Sevitsky. 2000. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience* 12, 14 (2000).
- [11] S. P. Reiss. 2009. Visualizing The Java Heap to Detect Memory Problems. In *5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '09)*.
- [12] Eclipse Foundation. 2018. Eclipse Memory Analyzer (MAT). <https://www.eclipse.org/mat/>
- [13] Oracle. 2018. The HotSpot Group. <http://openjdk.java.net/groups/hotspot/>
- [14] Oracle. 2018. VisualVM. <https://visualvm.github.io/>
- [15] Edward R. Tufte. 2007. *The Visual Display of Quantitative Information (2nd edition)*. Graphics Press, Cheshire, CT, USA.
- [16] Stef van den Elzen and Jarke J. van Wijk. 2013. Small Multiples, Large Singles: A New Approach for Visual Data Exploration. *Comput. Graph. Forum* 32 (2013).
- [17] Marc Weber, Marc Alexa, and Wolfgang Müller. 2001. Visualizing time-series on spirals. In *Infovis*, Vol. 1.
- [18] Markus Wening, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l Conf. on Managed Languages & Runtimes (ManLang '18)*.
- [19] Markus Wening, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proc. of the 10th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '19)*.
- [20] Markus Wening, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.
- [21] Markus Wening and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 9th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '18)*.
- [22] Richard Wettel and Michele Lanza. 2007. Visualizing Software Systems as Cities. In *4th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '07)*.
- [23] Vladimir Šor and Satish Narayana Srirama. 2014. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software* 96 (2014).

5.2 Memory Cities

This section includes two paper [326, 327] on how to visualize the evolution of memory trees over time in an engaging 3D visualization inspired by the software city metaphor.

Work-In-Progress Paper:

Markus Weninger, Lukas Makor, Hanspeter Mössenböck:

Memory Leak Visualization using Evolving Software Cities. In *Proceedings of the 10th Symposium on Software Performance, SSP 2019, Würzburg, Germany, November 4 - 6, 2019*.

Full Paper:

Markus Weninger, Lukas Makor, Hanspeter Mössenböck:

Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proceedings of the Working Conference on Software Visualization, VISSOFT 2020, Adelaide, Australia, September 28 - October 2, 2020* (moved online). - **Best Paper Award**

Artifact:

The tool has been successfully evaluated as an artifact at the VISSOFT 2020 conference. The artifact is available at [329], a video of the tool can be found at <http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.mp4>, and the instructions to use the tool can be found in Appendix A.

Memory Leak Visualization using Evolving Software Cities

Markus Weninger[⊗], Lukas Makor[△], Hanspeter Mössenböck[⊗]

[⊗] Institute for System Software, Johannes Kepler University Linz, Austria

[△] Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

Abstract

Memory leaks occur when no longer needed objects are unnecessarily kept alive. They can have a significant performance impact, possibly leading to a crash of the application in the worst case.

Most state-of-the-art memory monitoring tools lack visualizations of memory growth over time. However, domains such as software evolution and program comprehension have shown that graphically visualizing the growth and evolution of a system can help users in understanding and interpreting this growth.

In this paper, we present ongoing research on how to visualize an application’s *memory evolution* over time using the *software city* metaphor. While software cities are typically used to visualize static artifacts of a software system such as classes, we use them to visualize the dynamic memory behavior of an application. In our approach, heap objects can be grouped by arbitrary properties such as their types or their allocating threads. These groups are visualized as buildings arranged in districts, where the size of a building corresponds to the number of objects it represents. Continuously updating the city over time creates the feeling of an evolving city. Users can then identify and inspect those buildings, i.e., object groups, that grow the most.

We integrated our approach into AntTracks, a trace-based memory monitoring tool developed by us, to prove its feasibility.

1 Introduction

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). A memory leak occurs if objects that are no longer needed remain reachable from GC roots due to programming errors. For example, a developer may forget to remove objects from their (long-living) containing data structures. These objects cannot be reclaimed by the garbage collector and will therefore accumulate over time [15].

Most state-of-the-art memory monitoring tools do not use graphical means to visualize such a growth. Instead, they just take two heap snapshots, calculate the difference of the number of objects for every type, and display these difference values in a table. As it

has been shown in other domains such as software evolution and program comprehension [8], we think that users can also profit from software visualizations in the domain of memory monitoring.

In their work, Knight and Munro [1, 3] promoted the use of *metaphors* when developing software visualizations. Metaphors act as a *mapping from the concepts or artefacts required to be displayed to their graphical representation*. One such visualization metaphor are *software cities*. Wettel and Lanza [4] used software cities to visualize software systems, where buildings represent classes, grouped into districts based on their packages. The size of a building is determined by the classes’ number of attributes and number of methods. Steinbrückner and Lewerentz [7, 9] adopted and extended this idea by visualizing the development history of software systems using elevated city maps. Software cities have also been used in virtual reality environments to support program comprehension, as done by Fittkau et al. [10].

In this paper, we present ongoing research on how to use the software city metaphor to visualize memory monitoring data. Our goal is to ease the inspection of memory growth over time by providing interactive easy-to-interpret visualizations to users. To achieve this, our contributions encompass:

- a method to layout and visualize a heap state as a software city, see Section 2 and Section 3.
- techniques to visualize the evolution of memory over time as an evolving software city, see Section 4.
- a prototype implementation of our visualization approach in Unity 3D, see Figure 1.

2 Data

To visualize the memory evolution of an application, we need continuous information about the live heap objects. To obtain this information for a single point in time, most tools use heap dumps. However, continuously dumping the heap would incur too much run-time overhead, since the application is halted during the heap dump. Thus, we use the AntTracks VM [11, 12, 14], a virtual machine based on the Java Hotspot VM, to collect memory data.

From this data, we can reconstruct the heap state at every garbage collection point. For every heap object, a number of properties can be reconstructed, in-

cluding its address, type, allocation site, the thread that allocated it, and the heap objects it references.

3 Heap State Visualization

Heap objects can be grouped by a combination of their properties which results in a grouping tree [13]. Such a grouping tree is typically displayed in a tree table view, similar to the one shown in Table 1.

	Objects
- Heap	100,000
- Thread 1	80,000
Type A	70,000
Type B	10,000
+ Thread 2	10,500
...	

Table 1: A tree table view representing a heap state grouped by allocating threads and types.

Many tools also show other advanced metrics beside the number of objects and use features such as color encoding to highlight certain object groups. This can easily become overwhelming and hard to interpret for novice users. Thus, we present an approach to visualize a heap state as a software city.

3.1 Buildings and Districts

In the software city metaphor, artifacts are visualized as buildings that are arranged in districts, which can again be contained in other districts. In our case, buildings represent leaf nodes of a grouping tree, while inner tree nodes are represented as districts.

Districts are flat structures. Their area is sized to enclose all their contained districts and buildings. A building is a structure with a height and an area that depends on the number of objects its tree node represents. One of our goals was to achieve building sizes that represent more-or-less realistic building measures of real-world buildings. As preliminary formulas, we came up with $2 * \sqrt[4]{n_{objects}}$ units as the height and $\sqrt{n_{objects}}$ square units as the area for buildings. Mapping units to meters, the 70,000 objects of **Type A** from Table 1 would, for example, be represented as a building that has an area of about 264.5 square meters and a height of 32.5 meters. With adjusted formulas for height and area, the same approach could be used to visualize the city based on the number of bytes. Mixing metrics, such as using the number of objects for the area and the number of bytes for the height, is up to future work. For example, having very few very large arrays, this could result in extremely narrow buildings that are extremely tall if implemented carelessly, which would distort a realistic city feeling.

3.2 Layout

To layout the districts and buildings, we used the *squarified tree map* algorithm by Bruls et al. [2]. As explained in the previous section, every building has an area based on the number of objects it represents.

The squarified tree map algorithm tries to shape the area of each building as an approximate square, such that they can be laid out in a way that makes their districts again resemble squares.

4 Evolution Visualization

To visualize the memory evolution over a selected time window, we apply time traveling. According to Wettel and Lanza [6], time traveling is achieved by stepping back and forth through the history of a system while the city updates itself to reflect its current state. In our case, the history is the sequence of grouping trees generated at every reconstructed heap state in the selected time window.

4.1 Layout

It is *not* enough to visualize these grouping trees one after another. For example, buildings could be added or removed between two heap states. This would change the layout of districts and thus the position of buildings. This leads to the problem that users could hardly figure out if and which two buildings in two different heap states represent the same tree node.

To overcome this problem we apply *static position animation* [5], which creates a general city plan in which all buildings remain at the same position during the animation. To do so, all grouping trees are merged into a *meta tree*. Every node in this meta tree stores the maximum number of objects represented by the respective node at any time. Then, the layouting of the city happens *once* based on the values in this meta tree to reserve space for every building based on its largest possible area. Then, to visualize the heap at a certain point in time, buildings are centered in the space that has been reserved for them.

4.2 Memory Leak Investigation Mode

If a memory leak exists, typically certain object groups grow stronger than others. This is especially the case if the objects are grouped by type. To make it easier for users to identify those object groups, i.e., buildings, that grew the most, certain buildings are shown in solid mode, while the others have reduced opacity, as shown in Figure 1.

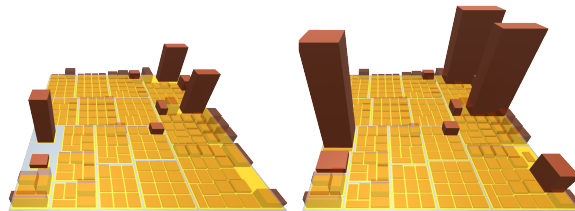


Figure 1: An application shortly after startup (left) and 60 garbage collections later (right). The ten buildings with the strongest growth are shown in solid mode, while the others have reduced opacity.

5 Interaction

Users can navigate through the city as a free-moving camera. The view can be tilted, rotated and zoomed by using the mouse wheel. By dragging the mouse or using the keyboard, the user can move the camera. Clicking on a building or district displays its information. This information includes the path from the tree root, e.g., *Overall Heap* → *Thread 1* → *Type A*, and the number of objects the structure represents.

To step back and forth in time, users are provided with buttons to go to the next and the previous heap state, as well as a slider to move through time. An automatic animation can also be played using a user-defined pause time between heap states. An example video of such an animation, showing AntTracks during trace file parsing, can be found here¹.

6 Conclusion and Future Work

In this paper, we presented an approach to visualize memory monitoring data using the software city metaphor. We discussed how a heap state, more specifically its heap objects, can be grouped into a tree, and how such a tree can be visualized as districts and buildings in a software city. Our approach is not only suitable for a single heap state, but can also visualize the memory evolution over time by using an advanced layout algorithm. Using our approach, the memory evolution of an application can be animated as a city that evolves over time, where growing buildings hint at an accumulation of objects that could be the result of a possible memory leak.

Since this work is still in progress, many possibilities exist for future work. Our current software cities only make use of three visual properties: area, height, and opacity. There is still potential for more advanced user interaction that may alter currently unused properties, such as letting users mark buildings of interest using custom colors. It is also interesting to explore how additional information such as object references could be included in a software city visualization. For example, selecting a building may also highlight / create visual links to other buildings that contain objects referenced by the selected building's objects. Also, a user study should be conducted to evaluate the usefulness of our new visualization approach.

7 Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

¹Video showing AntTracks as evolving memory city: <http://ssw.jku.at/General/Staff/Weninger/AntTracks/SSP19/MemoryCities-WorkInProgress.webm>

References

- [1] C. Knight and M. Munro. “Comprehension with [in] virtual environment visualisations”. In: *Int'l Workshop on Program Comprehension*. 1999.
- [2] M. Bruls, K. Huizing, and J. J. van Wijk. “Squarified Treemaps”. In: *Joint Eurographics and IEEE TCVG Symp. on Visualization*. 2000.
- [3] C. Knight and M. Munro. “Virtual but visible software”. In: *Conf. on Information Visualization*. 2000.
- [4] R. Wettel and M. Lanza. “Visualizing Software Systems as Cities”. In: *Int'l Workshop on Visualizing Software for Understanding and Analysis*. 2007.
- [5] G. Langelier, H. Sahraoui, and P. Poulin. “Exploring the evolution of software quality with animated visualization”. In: *Symp. on Visual Languages and Human-Centric Computing*. 2008.
- [6] R. Wettel and M. Lanza. “Visual Exploration of Large-Scale System Evolution”. In: *Working Conf. on Reverse Engineering*. 2008.
- [7] F. Steinbrückner and C. Lewerentz. “Representing Development History in Software Cities”. In: *Int'l Symp. on Software Visualization*. 2010.
- [8] R. Wettel, M. Lanza, and R. Robbes. “Software systems as cities: a controlled experiment”. In: *Int'l Conf. on Software Engineering*. 2011.
- [9] F. Steinbrückner and C. Lewerentz. “Understanding software evolution with software cities”. In: *Information Visualization 12.2* (2013).
- [10] F. Fittkau, A. Krause, and W. Hasselbring. “Exploring software cities in virtual reality”. In: *Working Conf. on Software Visualization*. 2015.
- [11] P. Lengauer, V. Bitto, and H. Mössenböck. “Accurate and Efficient Object Tracing for Java Applications”. In: *Int'l. Conf. on Performance Engineering*. 2015.
- [12] P. Lengauer et al. “Efficient Memory Traces with Full Pointer Information”. In: *Int'l. Conf. on Principles and Practices of Programming on the Java Platform*. 2016.
- [13] M. Weninger and H. Mössenböck. “User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring”. In: *Int'l Conf. on Performance Engineering*. 2018.
- [14] M. Weninger. *AntTracks*. 2019. URL: <http://mevss.jku.at/AntTracks>.
- [15] M. Weninger, E. Gander, and H. Mössenböck. “Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection”. In: *Int'l Conf. on Performance Engineering*. 2019.

Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor

Markus Weninger
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
markus.weninger@jku.at

Lukas Makor
CD Laboratory MEVSS
Johannes Kepler University Linz
Linz, Austria
lukas.makor@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract—Tool support is essential to help developers in understanding the memory behavior of complex software systems. Anomalies such as memory leaks can dramatically impact application performance and can even lead to crashes. Unfortunately, most memory analysis tools lack advanced visualizations (especially of the memory evolution over time) that could facilitate developers in analyzing suspicious memory behavior.

In this paper, we present *Memory Cities*, a technique to visualize an application’s heap memory evolution over time using the *software city* metaphor. While this metaphor is typically used to visualize static artifacts of a software system such as class hierarchies, we use it to visualize the dynamic memory behavior of an application. In our approach, heap objects can be grouped by multiple properties such as their types or their allocation sites. The resulting object groups are visualized as buildings arranged in districts, where the size of a building corresponds to the number of heap objects or bytes it represents. Continuously updating the city over time creates the immersive feeling of an evolving city. This can be used to detect and analyze memory leaks, i.e., to search for suspicious growth behavior. Memory cities further utilize various visual attributes to ease this task. For example, they highlight strongly growing buildings using color, while making less suspicious buildings semi-transparent.

We implemented memory cities as a standalone application developed in Unity, with a JSON-based interface to ensure easy data import from external tools. We show how memory cities can use data provided by AntTracks, a trace-based memory monitoring tool, and present case studies on different applications to demonstrate the tool’s applicability and feasibility.

Index Terms—Memory City, Software City, Software Map, Visualization Metaphor, Heap Memory Evolution, Memory Leak Analysis, 3D Visualization, Interactive Analysis System

I. INTRODUCTION

Modern programming languages such as Java use automatic garbage collection to free the developer from the error-prone task of allocating and freeing memory manually. To do so, heap objects that are no longer reachable from static fields or thread-local variables (so-called *GC roots*) are automatically reclaimed by a garbage collector (GC). Nevertheless, memory problems and anomalies such as memory leaks can still occur

The Memory Cities artifact (including binaries, data sets, video, and instructions) is available at [1], a video of the tool can be found at <http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.mp4>

This is the author’s version of the work. The definitive version was published in the Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT 2020).

<https://doi.org/10.1109/VISSOFT51673.2020.00017>

even in garbage-collected languages. For example, memory leaks happen if objects that are no longer needed remain reachable from GC roots. A common cause for this is that a developer accidentally forgets to remove objects from a (long-living) container data structure [2]–[4]. Such objects cannot be reclaimed by the garbage collector and will therefore accumulate over time.

To inspect a memory leak, users have to search for groups of objects that grow suspiciously over time. To perform such inspections, memory monitoring tools such as VisualVM [5] or Eclipse MAT [6] are often used. Unfortunately, many of these state-of-the-art tools do not use graphical means (except for time-series charts) to visualize the evolution of the heap. Yet, the usefulness of advanced visualization techniques in domains such as software evolution and program comprehension has already been shown in various user studies [7]–[14]. Thus, we think that developers can also profit from software visualizations in the domain of memory monitoring.

Visualizations often rely on metaphors to serve as “a mapping from the concepts or artefacts required to be displayed in the virtual world to their graphical representation” [15]. For example, in their inspiring works, Knight and Munro [15], [16] suggest the *software world* metaphor which consists of countries, cities, districts, and even details such as gardens and interior. Since this level of detail may not always be suitable or needed, the *software cities* metaphor emerged. Wetzel and Lanza [17], [18] used software cities to visualize the static structure of a software system, where *buildings* represent classes, grouped into *districts* based on their packages. The size of a building is determined by the classes’ number of attributes and number of methods. They extended this approach to also visualize the evolution of the code base over time [19]–[21]. Subsequent software city approaches used this metaphor to visualize the *dynamic* behavior of a software system based on recorded traces, such as *SynchroVis* to visualize concurrency [22] and *ExplorViz* to visualize the communication and dependencies between software components [11], [23]–[26].

Inspired by the widespread use of the software city metaphor, we combined existing techniques with new ideas to apply this metaphor to the domain of memory monitoring. In this paper, we present *Memory Cities*, an approach to visualize the heap evolution as an evolving software city.

In memory cities, buildings represent heap object groups that are arranged in districts based on shared heap object properties such as type. The size of the buildings can change over time, representing growing and shrinking heap object groups throughout the lifetime of a monitored application. Our goal is to ease the inspection and comprehension of memory growth over time, a common task in memory leak analysis, by providing an interactive and easy-to-understand visualization. Based on a work-in-progress report [27], this paper discusses the full *visualization pipeline* [28], [29] of memory cities, see Section III. In detail, our scientific contributions encompass:

- a data model based on which memory cities can be generated (Section IV),
- a discussion of the layout algorithm used (Section V),
- a mapping from memory metrics to visual attributes (Section VI),
- interaction features such as time traveling, information retrieval and a novel heap object reference analysis in 3D memory cities (Section VII),
- a fully functional 3D memory city visualization tool,
- various memory city case studies to showcase the approach’s feasibility and applicability (Section VIII).

II. BACKGROUND

Our memory city visualization has a well-defined JSON interface to be independent of a specific data source. Yet, to make this work more tangible, we regularly refer to data imported from the memory monitoring tool AntTracks [30].

Thus, this section presents the basics of AntTracks, a fully functional trace-based memory monitoring tool consisting of the *AntTracks VM* [31]–[33] and the *AntTracks Analyzer* [3], [4], [34]–[39].

A. Trace Recording by the AntTracks VM

The AntTracks VM (a slightly modified Java VM) records events such as object allocations and object movements performed by the GC during garbage collection and writes them into a trace file [31]. Additionally, the VM collects information about garbage collection roots and the references between objects [33], [37]. To reduce the trace size, the VM does not record any redundant data and applies compression [32].

B. Reconstruction in the AntTracks Analyzer

The AntTracks Analyzer incrementally processes the events in a trace file, reconstructing the heap state, i.e., the set of objects that were live in the monitored application, at every garbage collection point [34]. For every heap object, various properties can be reconstructed, including its memory address, type, allocation site, allocating thread, GC roots, the heap objects it references, and the heap objects it is referenced by.

The tool’s core mechanism is object classification and multi-level grouping [35], [36]. A classifier groups heap objects according to certain criteria such as type or allocation site. Grouping the heap objects based on multiple classifiers results in a hierarchical *grouping tree*. A common classifier combination is to group all heap objects by their types and then by

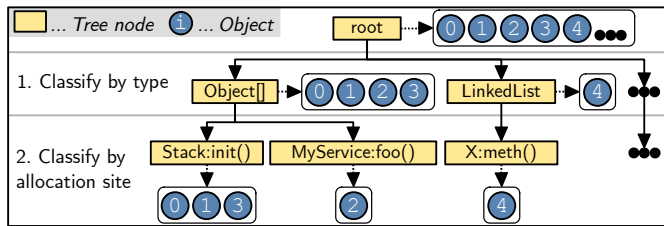


Fig. 1. A *classification tree* that first groups all heap objects by their types and then by their allocation sites.

their allocation sites, as shown in Figure 1. Yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object []`, of which the objects 0, 1 and 3 have been allocated in `Stack:init()` and object 2 has been allocated in `MyService:foo()`.

C. Common Techniques for Growth Visualization

As heap objects can be grouped by their properties, resulting in a grouping tree, it is common to display such data in a tree table, similar to the one shown in Table I.

TABLE I
A TREE TABLE VIEW REPRESENTING A HEAP STATE GROUPED BY TYPES AND ALLOCATION SITES.

	Objects
- Heap	100,000
- Type A	80,000
Allocated in foo()	70,000
Allocated in bar()	10,000
+ Type B	10,500
...	

Some tools also provide features to visualize the differences between two points in time. For this, they typically (1) take a heap snapshot at two points in time, (2) group the heap objects in both snapshots according to the same criteria, (3) calculate the differences of the number of objects for every tree node, and then (4) display these differences in a tree table. Even though object groups that grew between two points in time may hint at a memory problem, comparing two snapshots does not reveal general trends in an application’s memory behavior. To detect trends, the heap has to be compared at multiple points in time, a feature that is not supported by most state-of-the-art tools.

III. APPROACH

Our memory cities approach tackles the problem stated at the end of Section II-C: It aims to provide an intuitive and immersive visualization to inspect an application’s *heap evolution over time*. This section discusses the approach in general and presents its most important features and steps (shown in Figure 2) that also serve as an outline for the rest of the paper.

A. Overview

In general, a memory city displays a grouping tree, i.e., grouped sets of heap objects, as a 3D city visualization. Such a

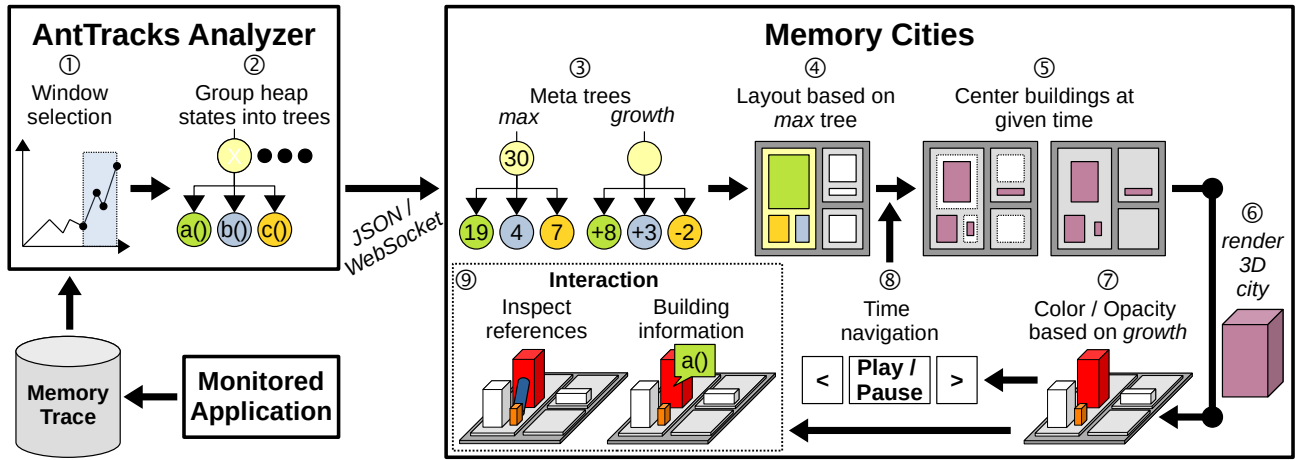


Fig. 2. Overview of our memory cities approach, corresponding to the typical visualization pipeline steps preprocessing, filtering, mapping, and rendering [29].

city consists of two types of structures: *buildings* and *districts*. Buildings represent tree *leaf* nodes, where a building’s area and height is determined by the number of objects / bytes represented by the respective tree node. For example, if the heap objects have been grouped by their types and allocation sites, each building represents a set of heap objects of the same type that have been allocated in the same method. These buildings are then grouped into districts based on their parent tree nodes, where districts can again be grouped into other districts. An example of such a city can be seen in Figure 3.

To generate such a layout, a tree map algorithm [40], [41] can be used. Figure 4 shows a tree map example, in which the orange parent node (district) represents 40MB of `Person` objects, with two yellow leaf nodes (buildings) representing 30MB allocated in method `m2` and 10MB allocated in method `m1`. As the set of classifiers that is used to group the heap objects is user-defined in AntTracks, various memory cities for different analysis purposes can be created. For example, if the user is interested in the most frequent types of objects allocated per thread, one could first group the heap objects by their allocating threads (districts) followed by their types (buildings). Memory cities can not only be inspected at a single point in time, but the user can step back and forth in time. This creates the feeling of an evolving city and enables users to search for strongly growing buildings, i.e., heap object groups that may be part of a memory leak. This task is further supported by the use of color highlighting and opacity settings.

B. Steps

Figure 2 presents the steps that lead from a recorded memory trace to the final memory city. The following list shortly describes each of them; they are explained in more detail throughout the rest of this work.

- ① Once a memory trace file has been loaded by the AntTracks Analyzer, the user sees the total heap memory utilization over time in a time-series chart. In this chart, the user can then select a suspicious time window, which may also be automatically suggested by AntTracks [39].
- ② Within the selected time window, the heap is grouped at

every garbage collection point according to a user-defined set of heap object properties, resulting in a list of grouping trees.

- ③ Based on these grouping trees, various *meta trees* are calculated. For example, a *max tree* stores the maximum number of objects and bytes a tree node represents at any point in time (in other words, the largest size a district or building may reach), while a *growth tree* stores the growth of each node between the first and the last grouping tree.

- ④ To reserve space for every building that will eventually be displayed in the city, we use the object/byte counts stored in the *max tree* as an input for the *squarified tree map* algorithm [42] to generate the city’s general layout *once*. By doing so, the generated layout ensures that every building could fit into the city even if all of them reached their largest size at the same point in time.

- ⑤ To display a memory city at a certain point in time, each building’s base area is calculated at that point and the building is then centered in the layout spot reserved for it.

- ⑥ Once every building has received its location, the building’s height is calculated and the corresponding cuboid is placed in the 3D environment.

- ⑦ To ease the search for growing structures, we use the growth information (stored in the *growth tree*) to highlight certain buildings using color and opacity.

- ⑧ The user can step back and forth through time to visualize the evolution of the city. When the user navigates between points in time, steps ⑤ to ⑦ are executed for each new point, and the visualization is updated. It is also possible to run this animation automatically to watch the whole city evolution without any user interaction needed.

- ⑨ Moving through time is not the only interaction possibility in memory cities. Users can also gather more information about a structure (i.e., a building or district) by hovering or clicking it. Another feature is to show references between two buildings. In the case of a memory leak, this feature helps users to differentiate between buildings, i.e., object groups, that cause other buildings to grow and those that grow because their object’s are kept alive by others.

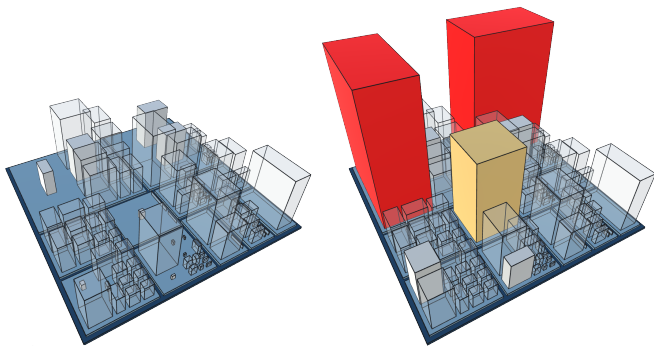


Fig. 3. An application’s heap visualized with memory cities shortly after startup (left) and 2 minutes / 300 garbage collections later (right). Districts are colored blue-ish based on their hierarchy level, buildings are colored from gray to red based on their growth. The ten buildings with the strongest growth are shown in solid mode, while the others have reduced opacity.

IV. DATA

This section discusses in more detail which data is needed by software cities in general, how this need translates to memory cities, and how we collect and process the needed data using AntTracks.

A. General

In general, a software city is built upon tree data. In its most basic form, each tree node contains a *key* for identification and at least one *value* based on which the city is laid out. Nevertheless, limiting each tree node to a single value also massively limits the number of visual attributes a software city can make use of. For example, a single value can be represented by the size of a building, with no other attributes such as color that could convey further information. If each tree node contained three values, one of them could be used to calculate a building’s base area, one could be used to calculate the building’s height, and one could be used to determine the building’s color, providing much more information for more diverse inspections. Using more visual attributes can make the visualization richer, yet complex mappings should be used for complex tasks or expert systems only since the mappings may become challenging to perceive [29]. Thus, when designing a new software city for a certain task (such as memory cities for the task of heap memory evolution analysis), the designers first have to decide whether they want to develop an expert system or a system that is also usable by novices.

B. Memory Cities

Since many expert memory monitoring tools already exist, our focus is to make memory anomaly inspection easier for novice users [43]. To achieve this, the goal of memory cities is to provide enough details to enable the detection of memory anomalies such as memory leaks, while keeping the visualization simple enough to understand it without prior training or explanations.

Once this decision is made, the next step is to define which data is needed. In general, a memory city is based on a

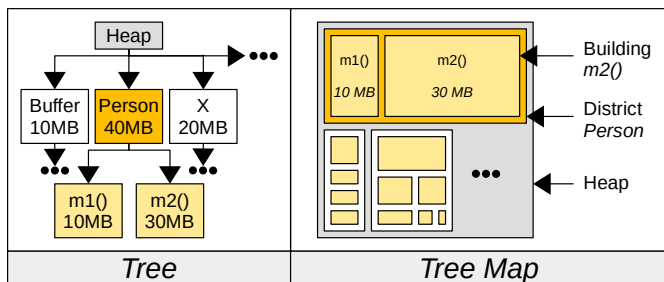


Fig. 4. We use tree mapping to lay out the buildings in memory cities.

tree in which each node represents a group of heap objects. As already discussed in Section II, a grouping tree can be constructed in AntTracks by applying a user-defined set of classifiers on the heap to group its objects accordingly. There are two ways to aggregate the heap objects in each node: Either by counting the number of objects, or by counting the number of bytes that the respective objects take up in the heap. We decided that for every tree node both metrics should be available for visualization in the memory city. Thus, our memory city tool expects the following data in each tree node:

- A unique *key* to identify the object group (e.g., “Heap#Person#m1”).
- A *name* to display (e.g., “m1 ()”).
- A *role* that specifies the object group’s grouping criteria (e.g., “Allocation Site”).
- An *object count* value.
- A *byte count* value.
- A *list of child nodes* which is empty for leaf nodes.

Since the heap grows and shrinks over time while an application is running (as new objects are allocated and others are freed by the GC), a major goal is to visualize this memory evolution. Especially, memory cities should help users to detect object groups that grow suspiciously strong, as this behavior hints at memory leaks. To this end, a memory city may not only load a single tree, but also a list of trees (representing heap states at garbage collection points), where each tree has a timestamp to ensure correct ordering.

Once such a list of trees has been imported, the memory city calculates two *meta trees* that are used to lay out the city and to highlight buildings: The *max tree* stores the maximum number of objects and bytes a tree node represents at any point in time (in other words, the largest size a district or building may reach), while a *growth tree* stores the growth of each node between the first and the last grouping tree.

Our memory city visualization has explicitly been developed to not depend on AntTracks’ grouping trees or any internals of AntTracks. To achieve this, we provide two ways of how to import data into our memory cities tool: Either by loading a list of grouping trees in JSON format¹ from disk, or by sending a list of grouping trees in JSON format to the memory cities tool via a WebSocket. Thus, any other memory monitoring tool besides AntTracks could also use our memory cities tool

¹JSON format example for list of grouping trees: <http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.json>

to visualize heap states and heap evolution, as long as the tool can provide a list of trees with the previously mentioned information per tree node.

V. LAYOUT

In the software city metaphor, artifacts are visualized as buildings that are arranged in districts, which can again be contained in other districts. In this section, we present how memory cities are laid out using the *squarified tree map algorithm* [42] and how we apply *static position animation* [44] to achieve a stable layout over multiple points in time.

A. Single Tree

In general, tree maps implicitly visualize a tree’s hierarchy via containment, i.e., the tree is visualized as a rectangle that contains other rectangles, which again can contain rectangles and so on. Thus, each rectangle represents a tree node, and the rectangle’s area is determined by one of the tree node’s values. In case of memory cities, this value is either the node’s object count or byte count. Instead of using the value directly (i.e., an increase of objects/bytes by a factor of 2 results in a building with a base area twice as big), a mapping function such as `sqrt` can be applied on the values beforehand.

To generate a tree map, we use a recursive algorithm that is given a tree node and a rectangle, which is then divided to fit the tree node’s child nodes [40], [41]. The alignment and rectangle ratio vary between different tree mapping algorithms [45]. We use the *squarified tree map* algorithm by Bruls et al. [42], which tries to shape the area of each tree node as an approximate square. This creates more realistic cities than using elongated shapes. The resulting layout is then used to generate the 3D city visualization by displaying leaf nodes as buildings and inner nodes as flat districts, which will be explained in more detail in Section VI.

B. Evolution Over Time

The visualization of the heap’s evolution over time, i.e., visualizing multiple trees one after another to inspect their growth, needs special handling, as it is *not* enough to perform a simple tree map layout whenever switching from one tree to another. One of the reasons for this is that if tree nodes were added or removed between two points in time, the respective rectangles in the tree map layout also have to be added or removed. This may happen if all objects of a certain type were collected by the GC, which leads to the disappearance of the respective tree node. Such a change in the tree structure would result in a change of the overall arrangement of districts and/or buildings. An unstable layout may cause users to lose track of a certain building. It becomes hard to figure out if and which two buildings in different heap states represent the same tree node, a core requirement for visualizations that want to visually express a system’s growth.

We apply *static position animation* [44] to overcome the problem of an unstable layout. This technique creates a general city plan in which all buildings and districts remain at the same position at every point in time. To create this general city plan,

we use the *max tree* presented in Section IV-B as an input to the tree map algorithm. Every node in this meta tree stores the maximum number of objects / bytes represented by the respective node at any time. This layout is calculated *once* when the memory city is initialized and contains a rectangle for every district and every building that will eventually be shown. More specifically, it reserves space for every district and building based on its *largest possible area*. Then, to visualize the heap at a certain point in time, buildings are centered in the space that has been reserved for them.

C. Tree Pruning

During the layout phase, it is also possible to prune the tree to reduce the complexity of the resulting memory city. For example, the tree map algorithm could be restricted to only take into account the N largest child nodes per parent, thus only reserving space for those buildings that represent larger groups of heap objects. When the city is shown for a certain point in time and no reserved space is found for a given tree node, this means that the object group is not relevant enough for the visualization and no building is shown for that node.

This feature is particularly useful for very wide trees. For example, grouping the live objects of a real-world application by type (e.g., `String`, `HashMap`, etc.) may result in hundreds or thousands of tree nodes, many of which may only represent a few objects [37]. Since one of the main goals of memory cities is to support the visualization of memory leaks, i.e., object groups that accumulate a large number of objects over time, small object groups are not of interest to the user and can be dropped. By default, memory cities have tree pruning enabled, using a user-defined number of child nodes to be shown.

VI. METRICS AND VISUAL MAPPING

As discussed in the previous section, the area of a building in a memory city depends either on the number of objects or the number of bytes its tree node represents. Yet, memory cities also use a number of other visual attributes to convey information to the user. This section discusses these attributes.

A. Districts

Similar to other software cities, districts in memory cities are flat structures, i.e., their height is fixed and does not encode information. Their purpose is to visualize the hierarchy of the underlying grouping tree. Thus, the bottom-most district always represents the whole heap, which may be divided into (multiple levels of) districts, one for each inner node in the underlying grouping tree. We use a linear *color gradient* from dark blue to light blue to encode a district’s level.

B. Buildings

As shown in Figure 5, in addition to the area metric (which is either based on the object count or the byte count a building represents) we further utilize the visual attributes height, color and opacity for each building. Each of these attributes will be discussed in the following.

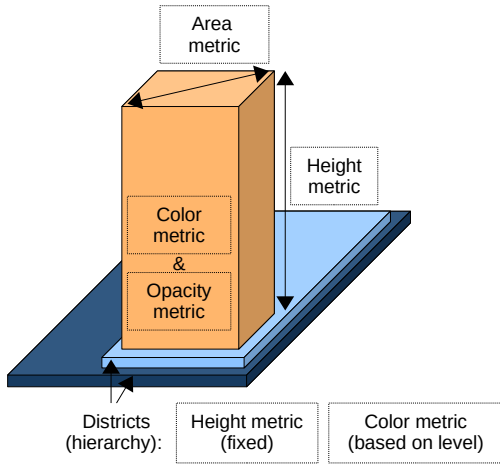


Fig. 5. Various visual attributes are used to express metrics.

1) *Height*: One of our goals was to achieve building sizes that represent more-or-less realistic measures of real-world buildings. Thus, for a building with an area of A square units, we use $2 * \sqrt{A}$ units as its height. This results in buildings that, if visualized with a perfectly squarified foundation, have a height twice the size of the building’s side length. Mapping units to meters, for example, a building with an area of 100 square meters (squarified side length of 10 meters) would have a height of 20 meters, while a building with an area of 400 square meters would have a height of 40 meters. Calculating the height based on the area means that both visual attributes represent the same metric, either object count or byte count. Mixing these metrics, i.e., using one metric for the area and the other one for the height, is still up to future research, since doing so did not yield satisfying results so far. For example, having a node that represents few very large arrays could result in (a) extremely narrow buildings that are quite tall (if the object count was used for the area and the byte count was used for the height) or (b) extremely wide buildings that are quite flat (if the byte count was used for the area and the object count was used for the height). Such unrealistic buildings would distort a realistic city feeling and would also be hard to interact with in certain situations (e.g., narrow tall buildings are hard to see and click). A possible solution in future work could be to use *categorical data* for the height, e.g., mapping the byte count to a few fixed heights such as tiny, small, medium, large and huge.

2) *Color*: Memory cities try to support users in understanding memory evolution (especially memory growth) over time. To make this task easier, memory cities encode the hitherto growth of a building as color. To this end, we utilize the *linear color gradient* shown in Figure 6.



Fig. 6. The color gradient used for buildings, ranging from gray (shrinking / no growth) over orange (medium growth) to red (strong growth).

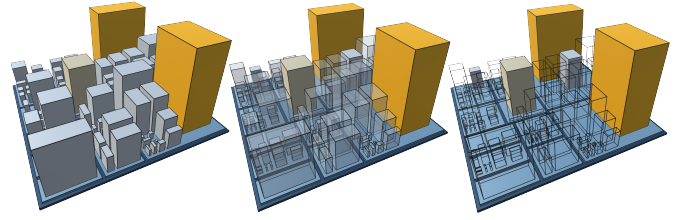


Fig. 7. Three different city representations. Left: Every building fully opaque. Middle: Five strongest growing buildings fully opaque, rest 40% opaque. Right: Five strongest growing buildings fully opaque, rest fully transparent.

The gradient maps a value in the range $[0, 1]$ to its respective color. Given a certain tree node with the identifier key , the access functions $first(key)$ and $cur(key)$ to query the node’s value (either objects or bytes) at the first point in time and at the current point in time, respectively, and the function $max()$ that returns the largest growth of any building stored in the *growth tree*. The color can then be calculated using $gradient((cur(key) - first(key))/max())$. Negative values are mapped to gray and represent buildings that shrank.

3) *Opacity*: To further increase the user’s focus on strongly growing object groups, the opacity of less important buildings can be decreased. The *growth tree* contains information about the growth between the first and the last point in time, i.e., the overall growth. Since object groups, i.e., buildings, that grew the strongest over the selected time window are those which are most likely involved in a potential memory leak, it seems reasonable to highlight those buildings and damp the others. Thus, memory cities allow the user to turn on the *building opacity* mode and select a number of N buildings that should stay opaque. As shown in Figure 7, the N buildings with the strongest growth (queried from the growth tree) stay fully opaque, while all other buildings are drawn at a user-defined reduced level of opaqueness (by default 40%). It is worth mentioning that the metric on which this visual attribute is based, namely the *overall* growth over the whole time window, differs from the metric used to define the building’s color, namely the *relative* growth since the start of the time window up to the current point in time. It is thus possible for a building to appear red and transparent at some point in time, i.e., strong growth up to that point but no strong overall growth, if the building shrinks again afterwards. Consequently, at the last point in time, those buildings that are shown opaque also have the most intense red color.

VII. INTERACTION

Users can navigate the camera through a memory city, they can step back and forth in time, they can click and hover structures to inspect them in detail, and they can display the number of references between buildings, i.e., heap objects. All of these features are explained in more detail in the following.

A. Navigation

The camera can be tilted, rotated and zoomed using the mouse wheel. By dragging the mouse or using the keyboard,

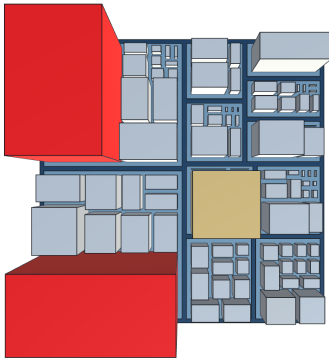


Fig. 8. The keyboard shortcut B positions the camera into a bird's eye view.

the user can move the camera. Memory cities also provide keyboard shortcuts for typical tasks. For example, pressing the B key moves the camera into a bird's eye view (see Figure 8), which can be useful to inspect the district structure.

B. Evolution Visualization: Time Travel

To visualize the memory evolution over time, we apply time traveling. Wettel and Lanza [19] define time traveling in the context of software cities as stepping back and forth through the history of a system while the city updates itself to reflect the current state. In our case, the history of the system is the sequence of grouping trees. The time stepping can be performed manually using buttons or a slider as well as using the arrow keys on the keyboard. Additionally, the evolution can also be animated automatically. During this animation, every heap state is shown for a user-defined period of time (0.5 seconds by default) before automatically switching to the next one. Users can pause and restart the animation at any point in time.

C. Structure Information

Hovering over a building or district displays information about its respective heap object group. This information includes the path from the tree root, e.g., *Heap* → *Type: Person* → *Allocation Site: foo()*, the number of objects and the number of bytes, as shown in Figure 9.

Besides showing a structure's information on hover, users can also click on a structure to highlight it, which is also shown in Figure 9. The structure stays selected when moving through time to make it easier to track its evolution.

D. Heap Object References

A novel feature of memory cities is the visualization of heap object references in a 3D environment. This feature is especially useful to reveal the root cause of a memory leak, since objects may accumulate over time even if they are not directly kept alive by a GC root, but rather indirectly by other objects, which would be the actual root cause of the problem. To fix such a leak, we want to find out the root cause by inspecting the references between the heap objects.

For example, imagine a memory leak caused by a `LinkedList<Person>` where persons are only added but

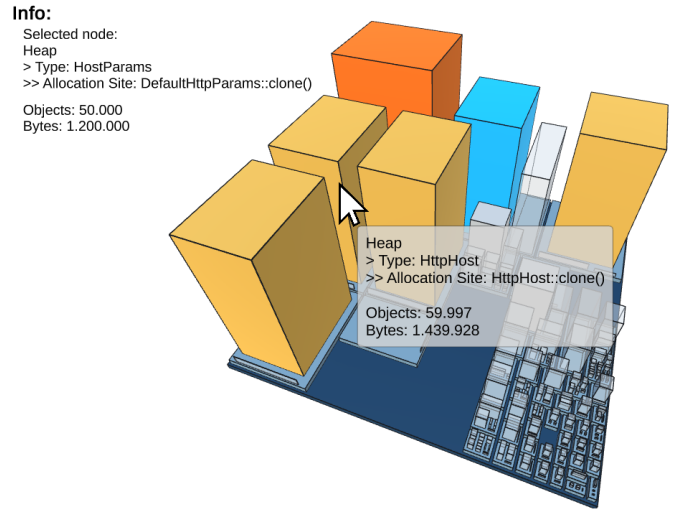


Fig. 9. Information about a structure is shown when hovering it (gray tool tip) or when selecting it with a click (selected building is highlighted in blue).

never removed. Further imagine that every person has a first name and a last name, each stored as a `String` field. Every addition to this list will result in six heap objects to be created: One `LinkedList$Node` that references the `Person` which in turn references two `String` objects which again reference a `char[]` each. ① in Figure 10 shows how such an application's memory city could look like if we group all heap objects by package (districts) and type (buildings). Since the application allocates more `String` and `char[]` objects than `Person` and `LinkedList$Node` objects, these two buildings are colored more intensively, even though they are only a *symptom* of the memory leak and not the *root cause*. To find the real root cause of the memory leak, we can inspect the references between the buildings. ② indicates that nearly all `char[]` instances are referenced by `String` objects (indicated by a thick purple frustum between the buildings). ③ shows the state of the memory city after selecting the `String` building. We can see that a lot of different types reference strings, but the most references come from `Person`, which is selected in ④. All persons are referenced by `LinkedList$Node` objects. ⑤ contains a very thin purple frustum which tells us that one of the nodes (i.e., the list head) is kept alive by the `LinkedList`.

To create the reference visualization, we utilize two maps, i.e., a *points-to map* and a *pointed-from map*, as shown in Table II. These maps (that, similarly to the grouping trees, can be imported as JSON files or via WebSockets) contain an entry for every building. For each building, they store how many objects the respective building references in other buildings, or by how many objects of other buildings it is referenced by, respectively. Based on these numbers and the building size itself, the frustums between the buildings can be sized, i.e., the more are objects involved, the bigger the visualized frustum. Since we know for every reference between two buildings how many objects are referencing and how many are referenced,

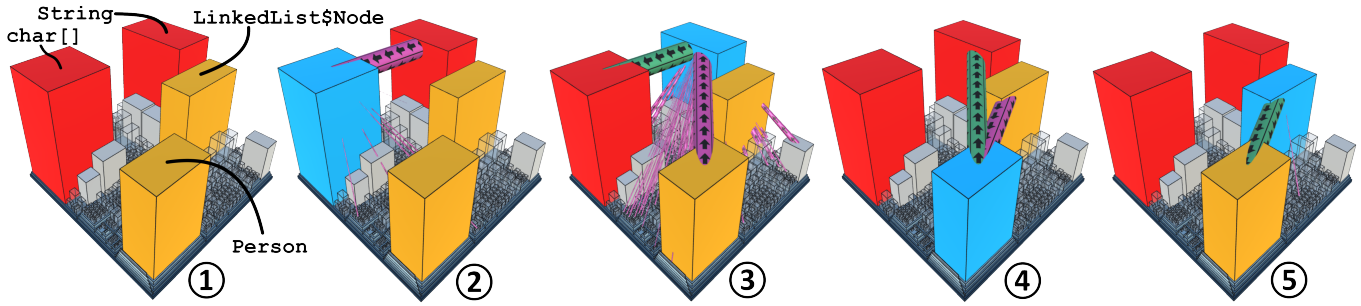


Fig. 10. Heap object reference analysis. The currently selected building in each step is highlighted in blue. Incoming references, i.e., references that keep objects alive in the selected building are colored purple. Outgoing references, i.e., references to objects that are kept alive by objects in the selected building are colored green. The more references there are between the objects of two buildings, the bigger the respective frustum.

TABLE II
A POINTED-FROM MAP AND A POINTS-TO MAP ARE USED AS DATA SOURCE TO CREATE THE REFERENCES BETWEEN BUILDINGS

Pointed-From Map		
String	Person	10,000
	Buffer	300
	...	
Person	LinkedList\$Node	10,000
...		
Points-To Map		
LinkedList	LinkedList\$Node	1
LinkedList\$Node	Person	10,000
Person	String	20,000
String	char[]	20,000
...		

we can even scale the start and the end radius of the frustum differently. For example, if 1% of the objects in building *A* reference 80% of objects in building *B*, the radius of the frustum attached to *A* will be much smaller than the radius at *B*, indicating a *few-to-many* reference. This information can especially be useful to detect (few) arrays that reference a lot of other objects. Vice versa, this technique can also indicate a *many-to-few* reference behavior, i.e., many objects share few other objects. Currently, a reference between two buildings is shown as a straight color-textured frustum, which might cut through other buildings in its way. Future research includes the evaluation of different reference placement techniques, for example pipe routing [46] or hierarchical edge bundling [47].

VIII. CASE STUDIES

To explain how memory cities can be used and to argue their usefulness and applicability, we present two case studies in which we use them to investigate memory leaks. To this end, we searched for real-world applications that contain memory leaks. In the following, we present the analysis of a memory leak in the *Commons HttpClient* library, as well as the analysis of a memory leak in the *Dynatrace easyTravel* application.

A. Commons HttpClient

Finding applications or libraries that contain memory leaks requires lots of effort, since their source code and the needed build tools have to be publicly available. To find the memory

leaking library discussed in this section, we browsed Apache’s issue tracker² for the keyword *leak*. This way, we found an old issue regarding a memory leak in the *Commons HttpClient* library, a library that can be used to send HTTP requests. As we did not know the library beforehand, it seemed like a good example to check if memory cities are helpful to detect proliferating objects even in an unknown application. We downloaded the affected version 3.0.1³ and built a small driver application⁴ which creates HTTP connections in multiple batches. In each batch, 10,000 connections are created and deleted shortly thereafter. One would expect to see spikes in the memory usage, as it should go up when connections are created and should go down after their deletion.

Contrary to this assumption, AntTracks reported a continuous memory growth in the application. Thus, we decided to inspect the heap evolution using memory cities. To do so, we selected the *type* and *allocation site* classifiers to be used at multiple GC points to generate grouping trees, which were then imported into the memory cities tool. The left half of Figure 11 shows the evolution of the resulting city over time. As we can clearly see in the third picture, six buildings grew strongly. Inspecting their type names and allocation sites, i.e., the methods in which the object have been allocated, already revealed interesting insights. In addition to that, the right-hand side of Figure 11 shows the reference patterns we observed. This made it clear that the leak has to do with *HostConnectionPool* objects that are kept alive (purple frustum) by *HashMap\$Node* (i.e., the nodes of a *HashMap*).

Knowing this reference pattern and the name of the method in which the accumulating *HostConnectionPool* objects are allocated provides us enough information to investigate the problem on the source code level. In the allocating method, we find that the *HostConnectionPool* objects are added to a map upon the creation of a new HTTP creation. However, they are not removed from that map when the connection is deleted, resulting in a memory leak.

²Apache’s issue tracker for HttpClient: <https://issues.apache.org/jira/projects/HTTPCLIENT/issues>

³Commons HttpClient in version 3.0.1: <https://mvnrepository.com/artifact/commons-httpclient/commons-httpclient/3.0.1>

⁴Driver application: <https://github.com/NeonMika/httpclient-leak-driver>

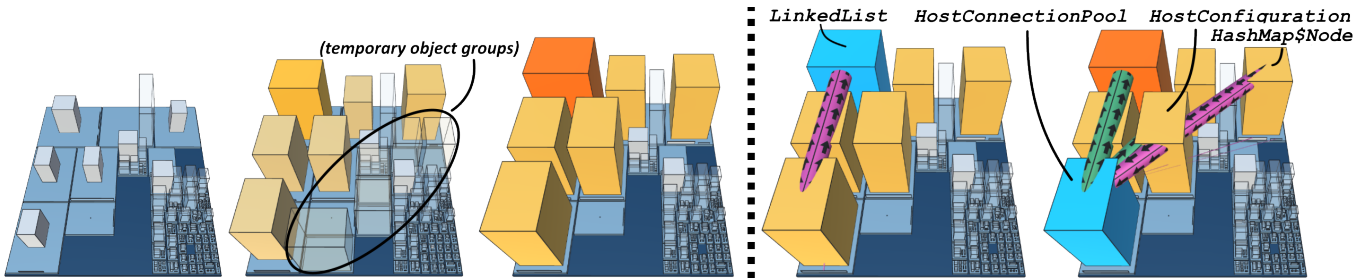


Fig. 11. In the Commons HttpClient application, `HostConnectionPools` (that reference `HostConfigurations` and `LinkedLists`) are kept alive because they are added to a `HashMap` but never removed.

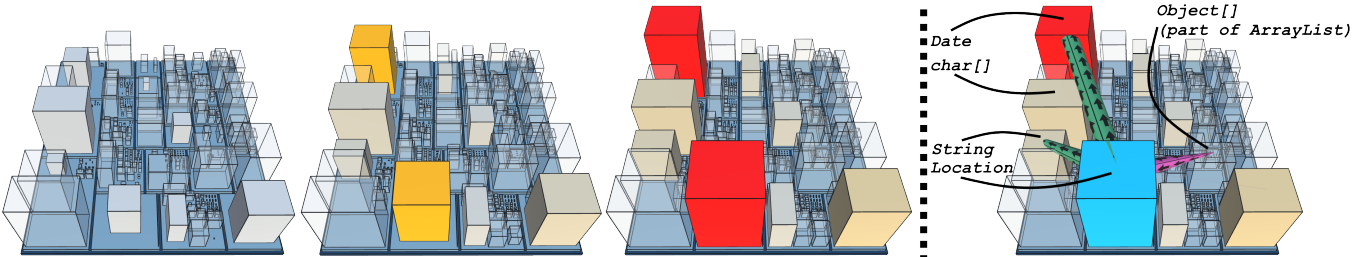


Fig. 12. In `easyTravel`, `Location` objects accumulate over time, together with many `Date` objects and a few `String` objects that they reference.

B. `easyTravel`

The second investigated application is *Dynatrace easyTravel*. Dynatrace focuses on application performance monitoring (APM) and distributes `easyTravel` as their state-of-the-art demo application. It is a multi-tier travel agency application, using a Java backend. A built-in load generator can simulate accesses to the service. When `easyTravel` is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

To inspect the heap evolution over time, we grouped all heap objects by *type* and *closest domain call site*, i.e., the method within `easyTravel` that led to the allocation even if the allocation itself was hidden inside a third-party framework. Figure 12 depicts the resulting memory city as it evolves over time. The two buildings that are clearly visible as strongest contributors to the heap growth represent `Location` and `Date` objects, each allocated by a certain method. To inspect if this parallel growth is coincidental or caused by either of the two, we inspected their references, as shown on the right-hand side of Figure 12. This makes it clear that the `Locations` reference the `Date` objects, as well as some `Strings`.

Using this information, we inspected the problem on the source code level. We found that the method in which all `Location` object are allocated is only called by the method `findLocations` in class `JourneyService`. There, we found a map that should have served as a cache for location searches. Once a search has been executed, a `QueryKey` instance is created and stored in the map, together with a list of the `Location` objects (the backbone of these lists can also be seen connected to the `Location` building via a purple frustum in the last picture of Figure 12). Subsequent searches for the same key should have found the respective entry in the map. However, `QueryKey` neither implements

`hashCode` nor `equals`. Thus, every request (even for an already existing key) resulted in a cache miss, which led to this typical memory leak.

IX. RELATED WORK

In this section, we discuss the use of visualization metaphors in general, as well as the application of the software city metaphor in various domains.

A. Using Visualization Metaphors

The use of metaphors in information visualization is widespread and has a long history. In general, metaphors such as *more is bigger* (e.g., bigger visual artifacts represent more of the underlying objects) or *similarity is closeness* (e.g., similar objects are positioned more closely to each other) often unconsciously shape the way we think and act [48]. In the following, we present a few examples of visualization that explicitly state the use of metaphors. For example, Waguespack [49] used geometrical figures as a metaphor for coding constructs to teach programming concepts. Boyle and Gray [50] used 3D structures to visualize database query results, using attributes such as size and position to convey information. More immersive and advanced usages of metaphors include colored virtual reality tunnels for program analysis and comprehension of concurrent programs [51], [52], or interactive map-like interfaces to visualize academic research fields and their similarity to each other [53].

B. Software Cities and Related Metaphors

As explained in Section I, Knight and Munro [15], [16] promoted the use of metaphors for software visualizations, especially their metaphor of a *software world*. As an alternative to software worlds, 3D city visualizations emerged. While

early 3D city visualization contained a lot of details and sophisticated layouts [54], most modern software cities are based on tree maps that have been extended to three dimensions [55]. New stable tree map algorithms [56], [57] may improve the process of laying out software cities in the future.

Software cities and similar metaphors have been applied in a variety of domains [29], [58]. For example, Langelier et al. [44], [59] as well as Bohnet and Döllner [60] used software cities to visualize quality metrics of software systems. Wettel and Lanza [8], [17]–[21] used software cities to visually explore the evolution of large-scale software system over time. Steinbrückner and Lewerentz [61], [62] adopted and extended this idea by visualizing the development history of software systems using elevated city maps. Software cities have been applied in the domains of concurrency visualization [22], software component communication and dependency visualization [11], [23]–[26], software performance visualization [63], [64], business process visualization [65], and test case analysis [66], [67]. Software cities have also been used in virtual reality [13], [14], [64], [68] and have been integrated into computer games such as Minecraft [69]. To the best of our knowledge, we are the first to employ the software city visualization metaphor in the domain of memory monitoring.

X. CURRENT LIMITATIONS AND FUTURE WORK

In this section, we discuss current limitations of our work and how we will address them in the future.

A. User Study

We believe that memory cities are a useful metaphor to inspect memory growth, especially for novice users that could otherwise be easily overwhelmed if the visualized data was presented in raw format or tables. We presented case studies to demonstrate the usefulness of memory cities and to showcase how they can be used to inspect real-world applications. Nevertheless, a more thorough evaluation is still missing. We thus plan to conduct a user study in the future to compare the performance of participants who use memory cities with the performance of those who use other tools.

B. Expert Mode

Currently, a primary focus of memory cities is to make the task of *memory leak analysis* more novice-friendly. For this, we rely on a small set of visual attributes, namely area, height, position, color, and opacity. In their taxonomy of software maps (the term software city is not uniquely defined in the software cartography domain), Limberger et al. [29] presented a large set of visual attributes that can be used to map data to the software city metaphor. However, they also mention that *a complex mapping [...] should be used for complex tasks or expert systems only*. Thus, we plan to further expand the feature set of memory cities in the future, including the use of more complex visual mappings such as a more advanced growth visualization using different object shapes and juxtaposition. These “expert mode features” should not be enabled by default but could be switched on by experienced

memory analysts. Memory cites can also be expanded to support other typical memory analysis tasks such as *memory churn analysis* [70], [71] or *memory bloat analysis* [72]–[76].

XI. CONCLUSION

In this paper, we presented our *memory cities* approach to visualize memory monitoring data using the software city metaphor. We discussed how a heap state, more specifically its heap objects, can be grouped into a tree, and how such a tree can be visualized as districts and buildings. Our approach is not only able to display a single heap state, but can also visualize the memory evolution over time by using static animation positioning and time traveling. Our approach can animate the memory evolution of an application as a city that evolves over time, where growing buildings hint at a proliferation of objects that could be the result of a possible memory leak. Such growing buildings are further highlighted using color and opacity.

We implemented our approach as a standalone 3D visualization tool using Unity and presented case studies on different applications to show its feasibility and usefulness. Memory cities have especially been designed with a focus on easy accessibility even for novice users. We hope that they can assist experienced users as well as users with a limited background in memory analysis to visually inspect their applications for memory anomalies and problems. We also think that memory cities and their immersive visualizations could even be used for other tasks besides typical memory analysis. For example, they could be used in software engineering education to teach students about the risks of careless use of memory in a less theoretical but more tangible way.

XII. ACKNOWLEDGEMENT

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

- [1] M. Weninger, L. Makor, and H. Mössenböck, “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor - Artifact (Binaries, Data Sets, Video, Instructions),” 2020. [Online]. Available: <http://doi.org/10.5281/zenodo.3991785>
- [2] G. H. Xu and A. Rountev, “Precise Memory Leak Detection for Java Software Using Container Profiling,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 17:1–17:28, 2013. [Online]. Available: <http://doi.org/10.1145/2491509.2491511>
- [3] M. Weninger, E. Gander, and H. Mössenböck, “Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring,” in *Proc. of the 9th Symp. on Software Performance (SSP)*, 2018, pp. 64–66. [Online]. Available: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf
- [4] —, “Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection,” in *Proc. of the 2019 ACM/SPEC Int’l. Conf. on Performance Engineering (ICPE)*, 2019, pp. 273–284. [Online]. Available: <http://doi.org/10.1145/3297663.3310297>
- [5] Oracle. (2020) VisualVM. [Online]. Available: <http://visualvm.github.io/>
- [6] Eclipse Foundation. (2020) Eclipse Memory Analyzer (MAT). [Online]. Available: <http://unity.com/>

- [7] B. Cornelissen, A. Zaidman, A. van Deursen, and B. V. Rompaey, "Trace Visualization for Program Comprehension: A Controlled Experiment," in *Proc. of the 17th IEEE Int'l. Conf. on Program Comprehension (ICPC)*, 2009, pp. 100–109. [Online]. Available: <http://doi.org/10.1109/ICPC.2009.5090033>
- [8] R. Wetzel, M. Lanza, and R. Robbes, "Software Systems as Cities: A Controlled Experiment," in *Proc. of the 33rd Int'l. Conf. on Software Engineering (ICSE)*, 2011, pp. 551–560. [Online]. Available: <http://doi.org/10.1145/1985793.1985868>
- [9] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller, "Comparing Trace Visualizations for Program Comprehension Through Controlled Experiments," in *Proc. of the 23rd IEEE Int'l. Conf. on Program Comprehension (ICPC)*, 2015, pp. 266–276. [Online]. Available: <http://doi.org/10.1109/ICPC.2015.37>
- [10] F. Fittkau, A. Krause, and W. Hasselbring, "Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment," in *Proc. of the 3rd IEEE Working Conf. on Software Visualization (VISSOFT)*, 2015, pp. 36–45. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2015.7332413>
- [11] —, "Software Landscape and Application Visualization for System Comprehension with ExplorViz," *Inf. Softw. Technol.*, vol. 87, pp. 259–277, 2017. [Online]. Available: <http://doi.org/10.1016/j.infsof.2016.07.004>
- [12] A. F. Blanco, J. P. S. Alcocer, and A. Bergel, "Effective Visualization of Object Allocation Sites," in *Proc. of the IEEE Working Conference on Software Visualization (VISSOFT)*, 2018, pp. 43–53. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2018.00013>
- [13] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza, "On The Use of Virtual Reality in Software Visualization: The Case of the City Metaphor," *Inf. Softw. Technol.*, vol. 114, pp. 92–106, 2019. [Online]. Available: <http://doi.org/10.1016/j.infsof.2019.06.007>
- [14] —, "The City Metaphor in Software Visualization: Feelings, Emotions, and Thinking," *Multim. Tools Appl.*, vol. 78, no. 23, pp. 33 113–33 149, 2019. [Online]. Available: <http://doi.org/10.1007/s11042-019-07748-1>
- [15] C. Knight and M. Munro, "Virtual but Visible Software," in *Proc. of the Int'l. Conf. on Information Visualisation, (IV)*, 2000, pp. 198–205. [Online]. Available: <http://doi.org/10.1109/IV.2000.859756>
- [16] —, "Comprehension with[in] Virtual Environment Visualisations," in *Proc. of the 7th Int'l. Workshop on Program Comprehension (IWPC)*, 1999, pp. 4–11. [Online]. Available: <http://doi.org/10.1109/WPC.1999.777733>
- [17] R. Wetzel and M. Lanza, "Visualizing Software Systems as Cities," in *Proc. of the 4th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2007, pp. 92–99. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2007.4290706>
- [18] —, "Program Comprehension Through Software Habitability," in *Proc. of the 15th Int'l. Conf. on Program Comprehension (ICPC)*, 2007, pp. 231–240. [Online]. Available: <http://doi.org/10.1109/ICPC.2007.30>
- [19] —, "Visual Exploration of Large-Scale System Evolution," in *Proc. of the 15th Working Conf. on Reverse Engineering (WCRE)*, 2008, pp. 219–228. [Online]. Available: <http://doi.org/10.1109/WCRE.2008.55>
- [20] —, "CodeCity: 3D Visualization of Large-Scale Software," in *Comp. Proc. of the 30th Int'l. Conf. on Software Engineering (ICSE Comp.)*, 2008, pp. 921–922. [Online]. Available: <http://doi.org/10.1145/1370175.1370188>
- [21] R. Wetzel, "Visual exploration of large-scale evolving software," in *Comp. of the 31st Int'l. Conf. on Software Engineering (ICSE Comp.)*, 2009, pp. 391–394. [Online]. Available: <http://doi.org/10.1109/ICSE-COMPANION.2009.5071029>
- [22] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring, "Synchrovis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency," in *Proc. of the 1st IEEE Working Conf. on Software Visualization (VISSOFT)*, 2013, pp. 1–4. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2013.6650520>
- [23] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring, "Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach," in *Proc. of the 1st IEEE Working Conf. on Software Visualization (VISSOFT)*, 2013, pp. 1–4. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2013.6650536>
- [24] F. Fittkau, A. van Hoorn, and W. Hasselbring, "Towards a Dependability Control Center for Large Software Landscapes," in *Proc. of the 10th European Dependable Computing Conf.*, 2014, pp. 58–61. [Online]. Available: <http://doi.org/10.1109/EDCC.2014.12>
- [25] F. Fittkau, P. Stelzer, and W. Hasselbring, "Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance," in *Proc. of the European Conf. on Software Architecture (ECSA)*, 2014, pp. 28:1–28:4. [Online]. Available: <http://doi.org/10.1145/2642803.2642831>
- [26] F. Fittkau, S. Roth, and W. Hasselbring, "ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes," in *Proc. of the European Conf. on Information Systems (ECIS)*, 2015. [Online]. Available: http://aisel.aisnet.org/ecis2015_cr/46
- [27] M. Weninger, L. Makor, and H. Mössenböck, "Memory Leak Visualization using Evolving Software Cities," in *Proc. of the 10th Symp. on Software Performance (SSP)*, 2019, pp. 44–46. [Online]. Available: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [28] S. dos Santos and K. Brodrie, "Gaining Understanding of Multivariate and Multidimensional Data Through Visualization," *Computers & Graphics*, vol. 28, no. 3, pp. 311 – 325, 2004. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0097849304000251>
- [29] D. Limberger, W. Scheibel, J. Döllner, and M. Trapp, "Advanced Visual Metaphors and Techniques for Software Maps," in *Proc. of the 12th Int'l. Symp. on Visual Information Communication and Interaction (VINCI)*, 2019, pp. 11:1–11:8. [Online]. Available: <http://doi.org/10.1145/3356422.3356444>
- [30] M. Weninger et al. (2020) AntTracks. [Online]. Available: <http://mevns.jku.at/AntTracks>
- [31] P. Lengauer, V. Bitto, and H. Mössenböck, "Accurate and Efficient Object Tracing for Java Applications," in *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2015, pp. 51–62. [Online]. Available: <http://doi.org/10.1145/2668930.2688037>
- [32] —, "Efficient and Viable Handling of Large Object Traces," in *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2016, pp. 249–260. [Online]. Available: <http://doi.org/10.1145/2851553.2851555>
- [33] P. Lengauer, V. Bitto, S. Fitzek, M. Weninger, and H. Mössenböck, "Efficient Memory Traces with Full Pointer Information," in *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*, 2016, pp. 4:1–4:11. [Online]. Available: <http://doi.org/10.1145/2972206.2972220>
- [34] V. Bitto, P. Lengauer, and H. Mössenböck, "Efficient Rebuilding of Large Java Heaps from Event Traces," in *Proc. of the Int'l. Conf. on Principles and Practices of Programming on The Java Platform (PPPJ)*, 2015, pp. 76–89. [Online]. Available: <http://doi.org/10.1145/2807426.2807433>
- [35] M. Weninger, P. Lengauer, and H. Mössenböck, "User-centered Offline Analysis of Memory Monitoring Data," in *Proc. of the 8th ACM/SPEC on Int'l. Conf. on Performance Engineering (ICPE)*, 2017, pp. 357–360. [Online]. Available: <http://doi.org/10.1145/3030207.3030236>
- [36] M. Weninger and H. Mössenböck, "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring," in *Proc. of the 2018 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2018, pp. 115–126. [Online]. Available: <http://doi.org/10.1145/3184407.3184412>
- [37] M. Weninger, E. Gander, and H. Mössenböck, "Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring," in *Proc. of the 15th Int'l. Conf. on Managed Languages & Runtimes (ManLang)*, 2018, pp. 14:1–14:13. [Online]. Available: <http://doi.org/10.1145/3237009.3237023>
- [38] M. Weninger, L. Makor, E. Gander, and H. Mössenböck, "AntTracks TrendViz: Configurable Heap Memory Visualization Over Time," in *Comp. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*, 2019, pp. 29–32. [Online]. Available: <http://doi.org/10.1145/3302541.3313100>
- [39] M. Weninger, E. Gander, and H. Mössenböck, "Detection of Suspicious Time Windows In Memory Monitoring," in *Proc. of the 16th ACM SIGPLAN Int'l. Conf. on Managed Programming Languages and Runtimes (MPLR)*, 2019, pp. 95–104. [Online]. Available: <http://doi.org/10.1145/3357390.3361025>
- [40] B. Johnson and B. Shneiderman, "Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures," in *Proc. of the IEEE Conf. on Visualization*, 1991, pp. 284–291. [Online]. Available: <http://doi.org/10.1109/VISUAL.1991.175815>
- [41] B. Shneiderman, "Tree Visualization with Tree-Maps: 2-D Space-Filling Approach," *ACM Trans. Graph.*, vol. 11, no. 1, pp. 92–99, 1992. [Online]. Available: <http://doi.org/10.1145/102377.115768>

- [42] M. Bruls, K. Huizing, and J. J. van Wijk, "Squarified Treemaps," in *Proc. of the Joint Eurographics and IEEE TCVG Symp. on Visualization (VisSym)*, 2000, pp. 33–42. [Online]. Available: http://doi.org/10.1007/978-3-7091-6783-0_4
- [43] M. Weninger, P. Grünbacher, E. Gander, and A. Schörgenhuber, "Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study," *Proc. ACM Hum.-Comput. Interact.*, vol. 4, no. EICS, Jun. 2020. [Online]. Available: <http://doi.org/10.1145/3394977>
- [44] G. Langelier, H. A. Sahraoui, and P. Poulin, "Exploring the Evolution of Software Quality with Animated Visualization," in *Proc. of the IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC)*, 2008, pp. 13–20. [Online]. Available: <http://doi.org/10.1109/VLHCC.2008.4639052>
- [45] W. Scheibel, M. Trapp, D. Limberger, and J. Döllner, "A Taxonomy of Treemap Visualization Techniques," in *Proc. of the 15th Int'l. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, 2020, pp. 273–280. [Online]. Available: <http://doi.org/10.5220/0009153902730280>
- [46] G. Belov, W. Du, M. G. de la Banda, D. Harabor, S. Koenig, and X. Wei, "From Multi-Agent Pathfinding to 3D Pipe Routing," in *Proc. of the Int'l. Symp. on Combinatorial Search (SOCS)*, 2020, pp. 11–19. [Online]. Available: <http://aaai.org/ocs/index.php/SOCS/SOCS20/paper/view/18513>
- [47] P. Caserta, O. Zendra, and D. Bodenes, "3D Hierarchical Edge bundles to Visualize Relations in a Software City Metaphor," in *Proc. of the 6th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, 2011, pp. 1–8. [Online]. Available: <http://doi.org/10.1109/VISSOF.2011.6069451>
- [48] G. Lakoff, *Master Metaphor List*. University of California, 1994.
- [49] L. J. W. Jr., "Visual Metaphors for Teaching Programming Concepts," in *Proc. of the SIGCSE Techn. Symp. on Comp. Sci. Ed.*, 1989, pp. 141–145. [Online]. Available: <http://doi.org/10.1145/65293.71203>
- [50] J. Boyle and P. M. D. Gray, "The Design of 3D Metaphors for Database Visualisation," in *Proc. of the 3rd IFIP 2.6 Working Conf. on Visual Database Systems*, vol. 34, 1995, pp. 185–202. [Online]. Available: http://doi.org/10.1007/978-0-387-34905-3_12
- [51] B. Reitinger, D. Kranzlmüller, and J. Volkert, "The MOST Immersive Approach for Parallel and Distributed Program Analysis," in *Proc. of the Int'l. Conf. on Information Visualisation (IV)*. IEEE Computer Society, 2001, pp. 517–522. [Online]. Available: <http://doi.org/10.1109/IV.2001.942105>
- [52] B. Reitinger, D. Kranzlmüller, and A. Ferko, "Program Visualization Through Visual Metaphors," in *Proc. of the Int'l. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2003. [Online]. Available: http://wscg.zcu.cz/wscg2003/Papers_2003/J79.pdf
- [53] A. Hiniker, S. R. Hong, Y. Kim, N. Chen, J. D. West, and C. R. Aragon, "Toward the Operationalization of Visual Metaphor," *J. Assoc. Inf. Sci. Technol.*, vol. 68, no. 10, pp. 2338–2349, 2017. [Online]. Available: <http://doi.org/10.1002/asi.23857>
- [54] T. Panas, R. Berrigan, and J. C. Grundy, "A 3D Metaphor for Software Production Visualization," in *Proc. of the Seventh Int'l. Conf. on Information Visualization (IV)*, 2003, pp. 314–319. [Online]. Available: <http://doi.org/10.1109/IV.2003.1217996>
- [55] T. Bladh, D. A. Carr, and J. Scholl, "Extending Tree-Maps to Three Dimensions: A Comparative Study," in *Proc. of the 6th Asia Pacific Conf. on Computer Human Interaction (APCHI)*, 2004, pp. 50–59. [Online]. Available: http://doi.org/10.1007/978-3-540-27795-8_6
- [56] W. Scheibel, C. Weyand, and J. Döllner, "EvoCells - A Treemap Layout Algorithm for Evolving Tree Data," in *Proc. of the 13th Int'l. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, 2018, pp. 273–280. [Online]. Available: <http://doi.org/10.5220/0006617102730280>
- [57] M. Sondag, B. Speckmann, and K. Verbeek, "Stable Treemaps via Local Moves," *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 729–738, 2018. [Online]. Available: <http://doi.org/10.1109/TVCG.2017.2745140>
- [58] P. Caserta and O. Zendra, "Visualization of the Static Aspects of Software: A Survey," *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 7, pp. 913–933, 2011. [Online]. Available: <http://doi.org/10.1109/TVCG.2010.110>
- [59] G. Langelier, H. A. Sahraoui, and P. Poulin, "Visualization-based Analysis of Quality for Large-scale Software Systems," in *Proc. of the 20th IEEE/ACM Int'l. Conf. on Automated Software Engineering (ASE)*, 2005, pp. 214–223. [Online]. Available: <http://doi.org/10.1145/1101908.1101941>
- [60] J. Bohnet and J. Döllner, "Monitoring Code Quality and Development Activity by Software Maps," in *Proc. of the 2nd Workshop on Managing Technical Debt (MTD)*, 2011, pp. 9–16. [Online]. Available: <http://doi.org/10.1145/1985362.1985365>
- [61] F. Steinbrückner and C. Lewerentz, "Representing Development History in Software Cities," in *Proc. of the ACM Symp. on Software Visualization (SOFTVIS)*, 2010, pp. 193–202. [Online]. Available: <http://doi.org/10.1145/1879211.1879239>
- [62] —, "Understanding software evolution with software cities," *Information Visualization*, vol. 12, no. 2, pp. 200–216, 2013. [Online]. Available: <http://doi.org/10.1177/1473871612438785>
- [63] K. Ogami, R. G. Kula, H. Hata, T. Ishio, and K. Matsumoto, "Using High-Rising Cities to Visualize Performance in Real-Time," in *Proc. of the IEEE Working Conference on Software Visualization (VISSOFT)*, 2017, pp. 33–42. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2017.25>
- [64] L. Merino, M. Hess, A. Bergel, O. Nierstrasz, and D. Weiskopf, "PerfVis: Pervasive Visualization in Immersive Augmented Reality for Performance Awareness," in *Comp. of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, 2019, pp. 13–16. [Online]. Available: <http://doi.org/10.1145/3302541.3313104>
- [65] S. Saito, "ProcessCity - Visualizing Business Processes as City Metaphor," in *Proc. of the CAISE Forum on Information Systems Engineering in Responsible Information Systems*, 2019, pp. 207–214. [Online]. Available: http://doi.org/10.1007/978-3-030-21297-1_18
- [66] A. Sosnowka, "Test City Metaphor as Support for Visual Testcase Analysis Within Integration Test Domain," in *Proc. of the Federated Conf. on Computer Science and Information Systems*, 2013, pp. 1353–1358. [Online]. Available: <http://ieeexplore.ieee.org/document/6644194/>
- [67] —, "Test City Metaphor for Low Level Tests Restructuration in Test Database," in *Proc. of the 8th Int'l. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2013, pp. 141–150. [Online]. Available: http://doi.org/10.1007/978-3-642-54092-9_10
- [68] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring Software Cities in Virtual Reality," in *Proc. of the 3rd IEEE Working Conf. on Software Visualization (VISSOFT)*, 2015, pp. 130–134. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2015.7332423>
- [69] G. Balogh and Á. Beszedes, "CodeMetropolis - A Minecraft based Collaboration Tool for Developers," in *Proc. of the IEEE Working Conf. on Software Visualization (VISSOFT)*, 2013, pp. 1–4. [Online]. Available: <http://doi.org/10.1109/VISSOFT.2013.6650528>
- [70] C. U. Smith and L. G. Williams, "Software Performance Antipatterns," in *Proc. of the Int'l. Workshop on Software and Performance (WOSP)*, 2000, pp. 127–136. [Online]. Available: <http://doi.org/10.1145/350391.350420>
- [71] M. Peiris and J. H. Hill, "Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern," in *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)*. ACM, 2016, pp. 237–248. [Online]. Available: <http://doi.org/10.1145/2851553.2851563>
- [72] N. Mitchell and G. Sevitsky, "The Causes of Bloat, the Limits of Health," in *Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007, pp. 245–260. [Online]. Available: <http://doi.org/10.1145/1297027.1297046>
- [73] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-scale Object-oriented Applications," in *Proc. of the Workshop on Future of Software Engineering Research (FoSER)*, 2010, pp. 421–426. [Online]. Available: <http://doi.org/10.1145/1882362.1882448>
- [74] G. H. Xu and A. Rountev, "Detecting Inefficiently-used Containers to Avoid Bloat," in *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2010, pp. 160–173. [Online]. Available: <http://doi.org/10.1145/1806596.1806616>
- [75] N. Mitchell, E. Schonberg, and G. Sevitsky, "Four Trends Leading to Java Runtime Bloat," *IEEE Software*, vol. 27, no. 1, pp. 56–63, 2010. [Online]. Available: <http://doi.org/10.1109/MS.2010.7>
- [76] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, pp. 23:1–23:50, 2014. [Online]. Available: <http://doi.org/10.1145/2560047>

5.3 Tree Visualizations

This section includes two papers [328, 330] on how to visualize the evolution of memory trees over time using the well-known tree visualization techniques sunburst and icicle.

Work-In-Progress Paper:

Markus Weninger, Lukas Makor, Hanspeter Mössenböck:

Heap Evolution Analysis Using Tree Visualizations. In *Proceedings of the 11th Symposium on Software Performance, SSP 2020*, Leipzig, Germany, November 12 - 13, 2020 (moved online).

Full Paper:

Markus Weninger, Lukas Makor, Hanspeter Mössenböck:

Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations. In *Proceedings of the Conference on Smart Tools and Applications in Graphics, STAG 2020*, Virtual Event, Italy, November 12-13, 2020. - **Best Paper**

Artifact:

The prototype of the visualization tool is available at <http://bit.ly/STAG-MemoryTreeVizTool>, a video of the tool can be found at <http://bit.ly/STAG-MemoryTreeVizVideo>.

Heap Evolution Analysis Using Tree Visualizations

Markus Weninger*, Lukas Makor*[⊗], Hanspeter Mössenböck*
{firstname.lastname@jku.at}

* Institute for System Software, Johannes Kepler University, Linz, Austria

[⊗] Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria

Abstract

Memory anomalies such as memory leaks can dramatically impact application performance and can even lead to crashes. Thus, supporting developers in understanding the heap memory behavior of their systems is essential. Unfortunately, most memory analysis tools lack advanced visualizations that could facilitate developers in analyzing suspicious memory behavior.

To analyze heap memory, it is common to group the heap’s objects, for example, by their types or by their allocation sites. Using multiple grouping criteria thus results in a tree-shaped representation of the heap content. Such a heap tree is then typically presented textually in a tree table.

In this paper, we present ongoing research on using well-known tree visualization techniques to visualize such heap trees as well as their evolution over time. Such visualizations may ease the detection of proliferating heap objects, facilitating memory leak analysis.

To demonstrate the feasibility and applicability of the presented approach, we implemented a web-based visualization tool and integrated it into AntTracks, our trace-based memory monitoring tool.

1 Introduction

Modern programming languages such as Java use garbage collection to automatically reclaim objects that are no longer reachable from static fields or thread-local variables. While this prevents a number of programming errors, certain problems such as memory leaks can still occur. For example, a developer may forget to remove objects from their containing long-living data structure. Consequently, these objects cannot be reclaimed by the garbage collector and thus accumulate over time [11].

Memory leaks cause more frequent garbage collections, which can have a significant negative performance impact. Worse, running out of memory even crashes the application. Thus, it is essential to provide tools to facilitate developers in detecting proliferating heap objects in their applications.

Even though *data visualization* [5] can help to convey information faster [6] and can aid in identifying patterns [7], most state-of-the-art memory monitoring tools do not take advantage of visualizations (except for time-series charts) and often present data in the form of tables and lists.

In this paper, we present work-in-progress to visualize memory evolution over time using tree visualizations. Our approach groups similar heap objects based on properties such as type, allocation site, or allocating thread into a *heap tree* (see Section 2). We then use well-known tree visualizations, i.e., the *sunburst plot* [1] and the *icicle plot* [9] to visualize the heap content at a single point in time (see Section 3). Generating heap trees at multiple points in time enables users to step through time to inspect the monitored application’s heap evolution over time (see Section 4). This helps users to recognize suspiciously growing object groups that may hint at memory leaks.

2 Data Collection

To inspect the heap at single point in time, most tools use heap dumps. Yet, to visualize the heap’s evolution over time, we need *continuous* information about its objects. Since regularly dumping the heap would incur too much run-time overhead (as the application is halted during the dump), we use the AntTracks VM [8], a modified Java virtual machine based on the Java Hotspot VM, to collect continuous memory traces (which introduces only around 5% run-time overhead [8]). We can reconstruct the heap state from such a trace at every garbage collection point. For every heap object, a number of properties can be reconstructed, including its address, type, allocation site, and the thread that allocated it. The heap objects can then be grouped by a user-defined combination of these properties which results in a *heap tree* [10].

3 Heap State Visualization

There is ample work on how to visualize tree-shaped data. Based on user studies that evaluated the usefulness [2] and aesthetics [3] of tree visualizations, we decided to use the sunburst plot as well as the icicle plot to visualize heap trees.

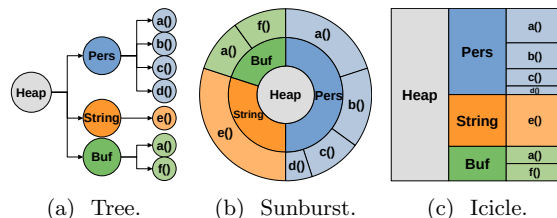


Figure 1: Three visualizations showing the same data.

3.1 Tree Visualizations

In a heap tree (Figure 1a), each tree node represents a group of heap objects. A good heap tree visualization should show the parent/child relationships as well as the number of objects/bytes represented by a specific node; the latter can be expressed as node size, but is often missing in simple tree visualizations. Thus, the *sunburst plot* (Figure 1b) as well as the *icicle plot* (Figure 1c) use variable-sized graphical elements to visualize nodes. The *sunburst plot* uses ring segments, where the angular size of the ring segments encodes a value. The *icicle plot* uses rectangles to encode a value using the rectangle’s height. Instead of using explicit links to depict the tree hierarchy, in the sunburst plot the tree hierarchy is moving outwards, starting at a *root circle* in the middle. Similarly, in the icicle plot the tree hierarchy is moving from left to right.

To compare these tree visualizations, Figure 1a through Figure 1c visualize the same data. Imagine that the underlying heap tree was generated by grouping all heap objects by their types, and all objects of the same type by their allocation sites. The gray root node (**Heap**) represents the whole heap. The nodes on the first level represent different types. For example, we can see that the heap consists of objects of the types **Pers** (blue), **String** (orange), and **Buf** (green). In the sunburst and icicle plot we can further see that 50% of the heap space is taken up by objects of type **Pers**. On the second level, allocation sites are shown. There we can see that objects of type **Pers** have been allocated at four different allocation sites, most of them at site `a()`.

3.2 Handling Huge Trees

Heap trees can be too *wide* or too *deep* to be visualized as a whole. For example, real-world applications use objects of hundreds of different types, thus grouping the heap objects by type would result in a tree with hundreds of siblings, i.e., a *wide* tree. On the other hand, using multiple grouping criteria may lead to a tree with lots of levels, i.e., a *deep* tree. Thus, we apply *tree pruning* to narrow trees and provide a *drill-down* feature to hide deep tree levels by default.

Tree Pruning To reduce a heap tree’s width, we only keep nodes that represent large object groups (i.e., those objects that most likely accumulated due to a memory leak), while smaller object groups are merged into artificial “*Other*” nodes. More specifically, we sort the child nodes of every node by their size (i.e., by their object count or byte count) and (1) keep the largest child nodes until they represent 90% of the objects on the current tree branch, yet we (2) keep a maximum of 9 child nodes. The remaining nodes are merged into an “*Other*” node.

Drill Down We only show two tree levels with the possibility to *drill down* into a certain tree branch. Clicking on a non-leaf node selects it as the new root

of the visualization. Figure 2 depicts an icicle that groups all heap objects by *allocating thread*, then by *type* and finally by *allocation site*. The left-hand side shows the icicle without drill-down (the allocation sites are not visible since only two levels are shown). The right-hand side shows the icicle after drilling down into the node *Thread 2*. To step out again, the user can click on the *Thread 2* root node.

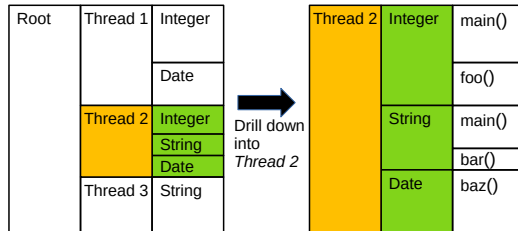


Figure 2: A drill-down feature enables users to explore deeper tree levels by selecting new tree roots.

4 Heap Evolution Visualization

It is not only possible to visualize the heap state at a single point in time, but also to visualize its evolution over time. For this, we use *time traveling* [4], a technique we already successfully applied in our *Memory Cities* visualization technique [12]. In time travelling, users can step back and forth through time, either using buttons or a time slider. After each step, the visualization updates itself to reflect the current heap state. Knowing which kinds of objects accumulate over time can greatly reduce the amount of source code that has to be inspected to fix a possible leak.

4.1 Stable Layout

A problem when switching from one point in time to another is that the order of the tree nodes could change. For example, if sibling nodes are ordered by size, and if their sizes change, the order of the nodes changes as well, which makes it hard to keep track of the evolution of different tree nodes.

In a *stable layout*, every node is assigned a sort position (based on a certain criterion) across all points in time *once* after all trees have been computed. This means that every node stays at the same relative position, e.g., at the second position, even if it grows or shrinks over time. In our heap evolution visualization, we currently sort all nodes based on their *end size*, i.e., based on their size in the last tree. For example, in Figure 3, the blue heap object group has the largest end size and is thus positioned first in both sunbursts.

4.2 Example

We implemented the presented approach as a `d3.js` web application¹. In Figure 3, we show a composition of tool screenshots taken while inspecting *DynaTrace easyTravel*, a state-of-the-art demo application

¹Prototype with example data hosted on <http://ssw.jku.at/General/Staff/Weninger/AntTracks/SSP20/WebTreeViz/>

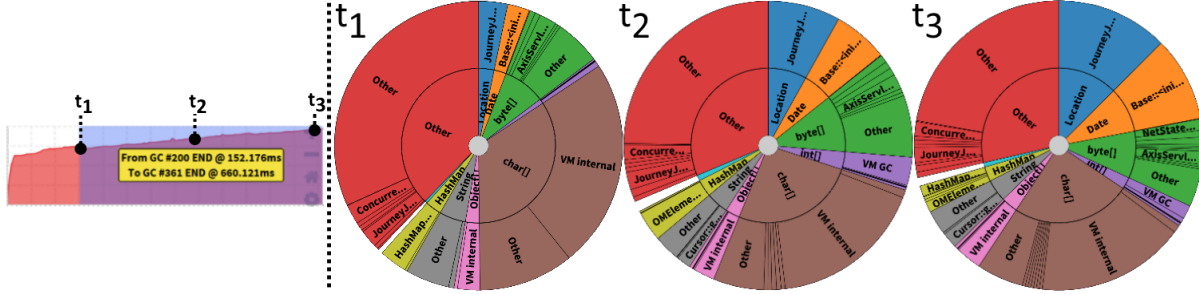


Figure 3: Heap evolution time travel through easyTravel shown at three different points in time t_1 , t_2 , and t_3 .

that simulates a broken travel agency website. As apparent in the time-series chart, easyTravel’s backend exhibits continuous memory growth. t_1 through t_3 exemplarily show our time travel sunburst visualization at three different points in time. The first tree level (inner ring) indicates the type, the second level depicts allocation sites. We can clearly see that the blue (type `Location`) and orange (type `Date`) segments grow. Both of these types are allocated only at a single allocation site each (as each type only has a single circle segment on the second level). Knowing which types accumulate the most objects (`Location` and `Date`) and where these objects are allocated (at a specific method in the class `JourneyJpaProvider`) makes it easy to locate the problematic code location.

5 Conclusion and Future Work

In this paper, we presented our approach to apply *tree visualizations* to facilitate heap memory analysis. We discussed how a heap state, more specifically its heap objects, can be grouped into a *heap tree* and how such a tree can be visualized using existing tree visualization techniques. We also visualize the heap evolution over time, where growing graphical elements hint at proliferating heap objects. These objects could be the result of a possible memory leak, which can then be inspected in more detail on the source code level based on information provided by the tree visualization.

Since this work is still in progress, various possibilities exist for future work. For example, our tool currently displays a single tree visualization, depicting the heap’s composition at a given point in time, and this visualization is updated when moving through time (a technique called *time traveling*). In the future, we plan to implement a *timeline view*. In this view, based on a number of points in time selected by the user, multiple tree visualizations are generated and displayed next to each other, similar to an interactive version of Figure 3. This should make it even easier for the user to detect growth trends.

6 Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

- [1] J. Stasko and E. Zhang. “Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-filling Hierarchy Visualizations”. In: *VISSOFT*. 2000, pp. 57–65.
- [2] S. T. Barlow and P. Neville. “A Comparison of 2-D Visualizations of Hierarchies”. In: *INFOVIS*. 2001, pp. 131–138.
- [3] N. Cawthon and A. V. Moere. “The Effect of Aesthetic on the Usability of Data Visualization”. In: *IV*. 2007, pp. 637–648.
- [4] R. Wetzel and M. Lanza. “Visual Exploration of Large-Scale System Evolution”. In: *WCRC*. 2008, pp. 219–228.
- [5] J. Heer, M. Bostock, and V. Ogievetsky. “A Tour through the Visualization Zoo”. In: *ACM Queue* 8.5 (2010), p. 20.
- [6] M. O. Ward, G. G. Grinstein, and D. A. Keim. *Interactive Data Visualization - Foundations, Techniques, and Applications*. A K Peters, 2010.
- [7] S. Murray. *Interactive Data Visualization for the Web*. O’Reilly Media, 2013.
- [8] P. Lengauer, V. Bitto, and H. Mössenböck. “Accurate and Efficient Object Tracing for Java Applications”. In: *ICPE*. 2015, pp. 51–62.
- [9] I. Bacher, B. M. Namee, and J. D. Kelleher. “Using Icicle Trees to Encode the Hierarchical Structure of Source Code”. In: *EuroVis*. 2016, pp. 97–101.
- [10] M. Weninger and H. Mössenböck. “User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring”. In: *ICPE*. 2018, pp. 115–126.
- [11] M. Weninger, E. Gander, and H. Mössenböck. “Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection”. In: *ICPE*. 2019, pp. 273–284.
- [12] M. Weninger, L. Makor, and H. Mössenböck. “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor”. In: *VISSOFT*. 2020.

Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations

M. Weninger¹ and L. Makor^{1,2} and H. Mössenböck¹

¹Johannes Kepler University Linz, Institute for System Software, Austria
²Johannes Kepler University Linz, Christian Doppler Laboratory MEVSS, Austria

Abstract

Memory leaks occur when no longer needed objects are unnecessarily kept alive. They can have a significant negative performance impact, leading to a crash in the worst case. Thus, tool support for heap evolution analysis, especially memory leak analysis, is essential. Unfortunately, most memory analysis tools lack advanced visualizations to facilitate this task.

In this paper, we present an approach to use well-known tree visualization techniques for memory growth visualization. Our approach groups heap objects into memory trees based on a user-defined set of properties such as their types or their allocation sites at multiple points in time. We present two novel approaches to inspect how these trees evolve over time: In our time-travel-based visualization, a single space-filling tree visualization shows the monitored application's heap memory at a given point in time. Users can step back and forth in time, causing the visualization to update itself. In our timeline-based visualization, a time-series chart depicts the overall memory consumption over time. Above this chart, multiple memory tree visualizations are shown side-by-side for a number of user-selected points in time. Using these techniques to visually inspect the evolution of the heap over time should enable users to gain new insights and to detect (problematic) memory trends in their applications.

To demonstrate the feasibility and applicability of the presented approach, we integrated it into AntTracks, a trace-based memory monitoring tool and applied it in two memory leak case studies.

CCS Concepts

•General and reference → Performance; •Software and its engineering → Software performance; Software maintenance tools; •Information systems → Data analytics; Information extraction; •Human-centered computing → Interactive systems and tools; Visualization techniques; Visual analytics; Information visualization;

1. Introduction

Many modern programming languages such as Java use a garbage collector (GC) to automatically reclaim heap objects that are no longer reachable from static variables or thread-local variables (i.e., GC roots). Even though automatic memory management prevents certain memory-related mistakes, various problems can still occur. Memory leaks are very common defects [GCS*20] and occur when objects remain reachable even though they are no longer needed. For example, a developer may forget to correctly clear a long-living data structure. Consequently, its objects cannot be reclaimed by the GC and thus accumulate over time [WGM18a, WGM19a].

Memory leaks can have a significant performance impact, leading to a crash in the worst case. Therefore, it is essential to provide tooling for memory analysis. Yet, existing tools have two major drawbacks: most of them (1) inspect the heap only at a single point in time and (2) do not use advanced visualizations. For example, existing tools such as VisualVM [Ora20] or Eclipse Memory Analyzer (MAT) [Ecl20] use heap dumps to inspect the heap at a single point in time. Yet, to detect and inspect trends in the memory

behavior, which is needed to investigate memory leaks, the heap has to be compared at multiple points in time [WGM19]. Those tools that do support memory evolution analysis often present the raw data in tables and do not employ visualization techniques. This is unfortunate, since domains such as software evolution and program comprehension have shown that using graphical means can help users in understanding and interpreting systems and their growth [CZvDR09, WLR11, FKH15, FFHW15, FKH17, BAB18].

In this paper, we tackle both of the mentioned problems by presenting an approach to visualize the heap memory evolution over time using tree visualizations. In order to create useful heap visualizations, the heap objects have to be brought into a suitable structure first. Many tools group heap objects based on a certain property (e.g., type or allocation site). Using multiple grouping criteria results in a hierarchical grouping structure, i.e., a memory tree [WLM17, WM18]. We record memory traces that allow us to create memory trees of a monitored application at multiple points in time, each representing the state of the heap at a certain point in time. In this work, we present a *time travel* visualization approach that enables users to step through the individual points

in time, as well as a *timeline* visualization approach that visualizes the heap at multiple points in time in juxtaposition side-by-side. This way, we enable users to recognize trends in the memory behavior of the monitored application to identify accumulating object groups, thereby gaining new insights that help to locate and resolve memory leaks. The contributions of this paper are:

- a suggestion of tree visualizations suitable for memory visualization based on a requirements catalog (Section 3).
- an approach to visualize a single heap state using the previously selected tree visualizations (Section 4).
- two novel approaches to visualize the evolution of the heap over time: the *time-travel-based* approach (Section 5.3) and the *timeline-based* approach (Section 5.4).
- an implementation of the presented techniques in the memory monitoring tool AntTracks (Section 6).
- two memory leak analysis case studies that demonstrate the approach’s feasibility and applicability (Section 7).

2. Background

This section explains how our approach collects memory data and how this data is transformed to be suitable for visualization. As the approach has been integrated into AntTracks, this section gives a short overview of the tool. AntTracks consists of two parts, the *AntTracks VM* [LBM15, LBF*16, LBM16], a virtual machine based on the Java Hotspot VM, and the *AntTracks Analyzer*, a trace-based memory analysis tool [WLM17, WM18]. We chose this tool since its source code is publicly available [Wen20] and the authors already had prior experience with its code base.

2.1. Trace Recording

While heap dumps are good enough to perform analyses at a single point in time, they fall short compared to continuous memory tracing approaches when performing analyses over time [WGM19a]. Thus, the AntTracks VM records memory events such as object allocations or objects moves during garbage collection and stores them in a trace file [LBM15]. This approach introduces a run-time overhead of about 5%, but provides more fine-grained memory information than heap dumps. To keep the size of the trace file low, the AntTracks VM tries to avoid storing redundant data [LBM16].

2.2. Heap State Reconstruction and Memory Trees

The AntTracks Analyzer uses the recorded trace file as input to reconstruct the memory data and provides various features to analyze this data. By incrementally parsing the recorded trace file, the tool is able to reconstruct a heap state for each garbage collection point [BLM15]. A heap state is a set of heap objects that were live in the monitored application at a certain point in time. Properties such as the address, type, allocation site, and allocating thread can be reconstructed for each heap object.

One of the core features of the AntTracks Analyzer is to perform object classification and multi-level grouping [WLM17, WM18]. This approach groups the heap objects into a hierarchical memory tree based on a user-defined set of criteria (called *classifiers*), e.g., based on their types, allocation sites, or allocating threads. Every

node in such a tree represents a set of objects that share the same properties, also called an object group. An exemplary memory tree created using the *allocating thread* classifier and *type* classifier is depicted in Figure 1. The root node of the memory tree represents the whole heap, every node represents an object group. For example, the node *T1* represents objects 0, 1 and 2, which were allocated by thread T1. The *Integer* node below *T1* represents objects 0 and 1, which were allocated by thread T1 and are of type *Integer*.

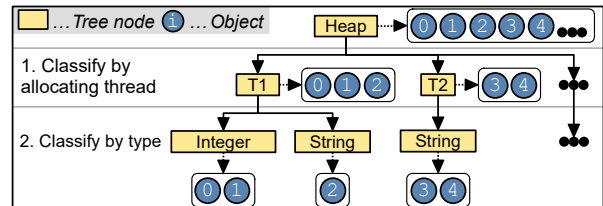


Figure 1: A memory tree that resulted from first grouping all objects by their allocating threads and then by their types.

3. Tree Visualizations

Memory traces enable reconstructing a vast amount of information about all heap objects that have been live at some point in time in the monitored application. Since presenting this data with raw numbers would most probably overwhelm the user, we chose tree visualizations as a means of data visualization [HBO10]. Data visualization can help to convey information faster [KK91, WGK10] and can facilitate the identification of patterns [War13, Mur13] which can lead to new insights [War13]. This section discusses different properties of tree visualization and presents a requirements catalog we used to select two tree visualizations that seem to be adequate for an interactive heap evolution visualization.

3.1. Tree Properties and Requirements Catalog

Tree visualizations are the most common type of visualization to depict hierarchies such as memory trees [SZ00]. Consequently, ample research has been performed on tree visualizations, which resulted in a vast number of tree visualization techniques. For example, *treevis.net* [Sch11] lists more than 300 publications on tree visualizations. However, not all tree visualization techniques are suitable for visualizing the heap memory evolution over time.

In general, different tree visualizations use different approaches to display a tree’s *content information* as well as its *structural information*. *Content information* associates data of the underlying tree nodes to visual attributes of the nodes (such as node size, color, or transparency). *Structural information* concerns the tree’s hierarchy and can either be expressed *explicitly* or *implicitly* [JS91]. Explicit visualizations, also called *node-link visualizations*, use explicit graphical elements such as lines between nodes to indicate relationships. Implicit visualizations, also called *space-filling visualizations*, indicate the relationships of the nodes via spatial arrangement, e.g., containment [SS06]. Various works provide further and more detailed taxonomies on tree visualizations [Sch11, STLD20].

To identify tree visualizations suitable for memory evolution visualization, we define four requirements that are vital to help users in gaining insights into the heap’s evolution over time.

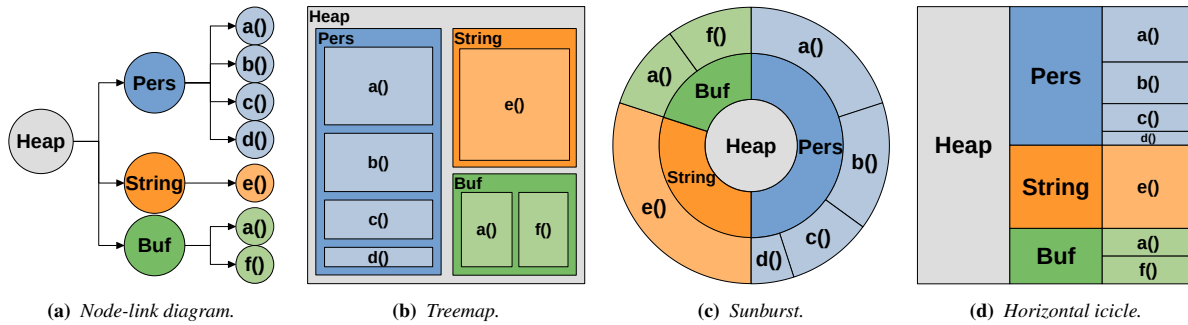


Figure 2: Four different tree visualization techniques depicting the same memory tree.

1. A node's content information (e.g., its number of heap objects) should at least be represented by size, following the visualization metaphor *more is bigger* [Lak94, HHK*17].
2. Updating hundreds of nodes should be feasible in real-time since our approach revolves around an interactive visualization. It must not involve inconvenient latency since this complicates making observations and drawing generalizations [LH14]. Furthermore, it has been shown that users tend to use an interactive system less often if it exhibits interaction latency [Bru09].
3. Since we want to make the tool accessible to novice users, the visualization has to be easy to understand, even for inexperienced users. Explicit explanation should not be necessary. Thus, we do not search for novel and experimental visualization techniques, but rather want to use visualizations that have been empirically proven to be effective, efficient and easy to use.
4. The tree visualization has to support a stable layout of evolving data over time. As Hahn et al. [HTMD14] state *layout stability is considered essential for [...] visual analysis tasks such as comparing hierarchies and attributes of such hierarchies' nodes, and tracking changes to hierarchies over time*, which are common tasks in memory analysis. When visually exploring data, users create *cognitive maps* that are based on spatial relations and attributes of the presented data [Kit94]. Consequently, we look for visualizations that support a stable layout, i.e., a layout that requires few changes to the user's cognitive map as the spatial relations mostly stay intact when updating the visualization.

3.2. Exemplary Tree Visualizations

Figure 2 shows four different tree visualizations, all of which depict the same tree. This memory tree was generated by grouping all heap objects by their types, and all objects of the same type by their allocation sites. The gray `Heap` root node represents the whole heap, nodes on the first level represent different types. We can see that the heap consists of objects of the types `Pers` (blue), `String` (orange), and `Buf` (green). In Figure 2b through Figure 2d we can further see that 50% of the heap is taken up by objects of type `Pers`. On the second level we can see that the `Pers` objects have been allocated in four different methods, most of them in method `a()`.

3.3. Selection of Tree Visualization Techniques

Most *explicit* visualizations (such as the node-link diagram in Figure 2a) do not use variable-sized nodes, making it hard to distin-

guish nodes that represent few or many objects. As this contradicts our first requirements, no explicit visualizations were chosen for our approach.

We also excluded visualizations that require complex layout calculations, as they cannot be updated fast enough to not disturb the users during analysis (requirement three). Examples for excluded visualizations encompass *variational circular treemaps* [ZL15] or *GosperMaps* [AHL*13], since calculating their layout based on a few hundred nodes already takes seconds.

To fulfill the third requirement, we explored existing study results. Barlow and Neville [BN01] performed two experiments to compare the performance of icicle visualizations, tree ring visualizations (a visualization similar to the sunburst visualization) and treemap visualizations. They found that icicle and tree ring both worked quite well, while treemap worked significantly worse than the other tree visualizations. Cawthon and Moere [CM07] conducted an online survey to evaluate the aesthetics and task performance of eleven visualization techniques. With regard to aesthetics, sunburst was the clear winner, while in terms of correctness and response time, both sunburst and icicle were among the best techniques. Treemap again ranked among the worst techniques.

The fourth requirement, i.e., that the chosen visualizations have to support stable layouts, is discussed in more detail in Section 5.1.

As icicle and sunburst fulfill all our requirements and consistently ranked among the best tree visualization techniques in the discussed studies, both were chosen to be used for our heap evolution analysis. Consequently, we chose to not include treemaps due to their negative study results. Nevertheless, treemap algorithms are still useful. For example, they are successfully used to generate layouts for *software maps* and *software cities* [LSDT19] such as *CodeCity* [WL07, WLR11], *SynchroVis* [WWF*13], *ExplorerViz* [FKH17], or *Memory Cities* [WMM19, WMM20].

3.4. Chosen Tree Visualizations

This section shortly explains the *sunburst* and *icicle* visualizations that were chosen to be part of our heap visualization approach.

Sunburst As shown in Figure 2c, sunburst is a radial space-filling visualization [SZ00]. In a sunburst, the root node of the hierarchy is depicted as a circle in the center of the visualization. This circle is surrounded by multiple levels of circular ring segments where each

ring segment represents a tree node. The tree hierarchy is moving outwards, i.e., each tree level is further away from the center. The sunburst visualization is adjacency-based, meaning that the children of a node are positioned next to each other on the next level within the angular sweep of their parent. The angular size of the segment is based on an attribute of that node, in our case the number of objects or bytes of the respective heap object group.

Icicle Icicle is another space-filling visualization [HBO10]. Similar to sunburst, it is adjacency-based, meaning that the children of each node are positioned next to each other on the next level within the extent of their parent. In a horizontal icicle [BNK16] (as shown in Figure 2d), each rectangle has the same width and its height is based on some attribute of its respective tree node.

4. Heap State Visualization

A memory tree, i.e., the result of grouping heap objects based on common properties, is the basis for our heap state visualization. Figure 3 shows an example of the same memory tree being displayed in our tool, once as an icicle and once as a sunburst.

Problems that can arise are that memory trees can be too *wide* or too *deep* to be visualized as a whole. For example, real-world applications use objects of hundreds of different types, thus grouping the heap objects by type would result in a tree with hundreds of siblings, i.e., a *wide* tree. On the other hand, using multiple grouping criteria may lead to a tree with lots of levels, i.e., a *deep* tree. Thus, we apply *tree pruning* to narrow trees and provide a *drill-down* feature to hide deep tree levels by default. This section discusses these techniques in more detail.

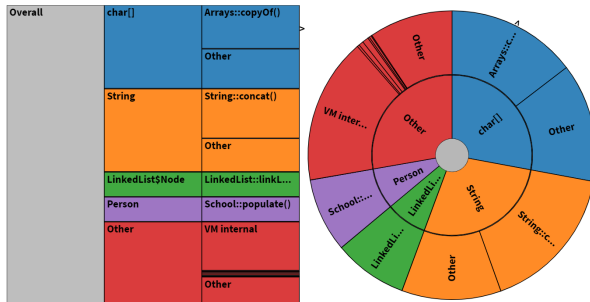


Figure 3: Screenshots of a memory tree visualized in our tool, once as an icicle and once as a sunburst.

4.1. Tree Pruning

The goal of tree pruning is to keep only the most significant nodes and to hide less important ones. In memory leak analysis, we are interested in nodes that represent large object groups, as their objects potentially accumulated due to a memory leak. Thus, unimportant smaller object groups can be merged. In our implementation, we sort the child nodes of every node by their size (i.e., by their object count or byte count) and (1) keep the largest child nodes until they represent 90% of the objects on the current tree branch, yet we (2) keep a maximum of 9 child nodes. The remaining nodes are merged into an artificial “Other” node.

4.2. Drill-Down

As the screen space is limited, it is neither feasible nor reasonable to display the full hierarchy of deep trees. Consequently, we decided to only display two levels below the root node by default, with the possibility to *drill-down* into deeper tree branches. By clicking on a non-leaf node, the selected node becomes the new root of the visualization. Figure 4 depicts an icicle that was created using the *allocating thread*, *type* and *allocation site* classifiers. On the left, the visualization is shown without drill-down. On the right, the drill-down has been performed on the node *Thread 2*. The orange and green rectangles highlight the node selected for drill-down as well as its children. Additionally, the allocation sites of the objects that were allocated by *Thread 2* are now shown in the drilled-down view. While in a drill-down, the user can click on the root node to step up one level again.

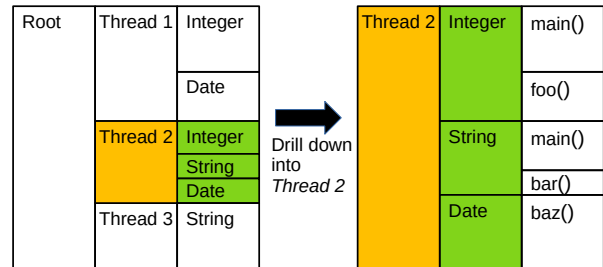


Figure 4: Drilling down into a node in the initial icicle (left) results in the selected node becoming the root of the visualization (right).

4.3. Local View and Global View

Showing only a limited number of tree levels enables us to display the shown nodes with reasonable size. Thus, also more room is available for text within the nodes (e.g., type names or method names). However, limiting the number of shown levels comes at the cost of losing the hierarchy overview, e.g., how many levels really exist. Users may also possibly lose track of their current drill-down position within the hierarchy after multiple drill-down steps, putting potential additional cognitive load on the user [TM04].

To tackle these problems, we use two synchronized visualizations next to each other. The first one, called the *local view*, only displays the currently selected node and its two sublevels (as discussed before). Additionally, a second view called the *global view* displays the full hierarchy, independent of the tree depth or the currently selected drill-down node. An example for this can be seen in Figure 5, where an icicle is shown that was generated using the *allocating thread*, *type* and *allocation site* classifiers. The *local view* on the left shows the *Thread 2* node (that has been selected via drill-down) as the root as well as its two direct sublevels. The *global view* on the right shows the full hierarchy, i.e., all tree levels, with the drill-down node highlighted. The other tree branches are set to semi-transparent. Having these two visualizations next to each other solves the problems of losing track in the overall hierarchy. To further make orientation easier, in addition to highlighting the currently selected node, when the user hovers a node in the local view we also highlight the respective node in the global view. This makes it easier to spot its position within the hierarchy. Local and global view (including an example) will be revisited in Section 5.3.

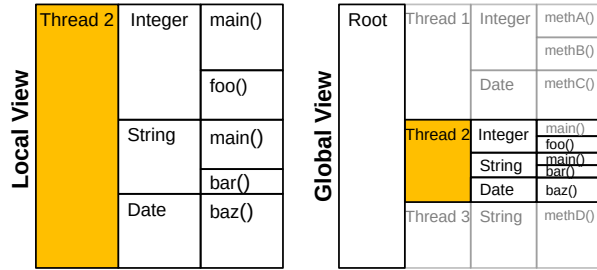


Figure 5: The local view (left) shows the current drill-down node as root and two more levels, while the global view (right) shows the whole hierarchy with highlighted drill-down branch.

5. Heap Evolution Visualization

Visualizing the heap evolution means to not only visualize the state of the heap at a single point in time, but to visualize its evolution across multiple points in time. To narrow down the search space, users can select a time window of interest for visualization [WGM19b]. Within the selected time window, a user-selected list of classifiers is used to group all live heap objects into memory trees at multiple points in time. This results in a sequence of memory trees, where each tree represents the heap state of the monitored application at a certain point in time. In this section, we discuss preprocessing steps to achieve a stable layout of the tree visualizations across multiple points in time, as well as a technique to better visualize absolute growth. Following, we present two novel approaches to inspect the evolution of these trees: the *time-travel-based* approach and the *timeline-based* approach.

5.1. Stable Layout

Visualizations that show the evolution of a system have to be carefully designed. A major risk is that a small change in the underlying data can result in vastly different layouts being generated. For example, between two points in time it can happen that the order of the tree nodes would change. Imagine two sibling nodes that are ordered by size: when their size, e.g., heap object count, changes, the order of their nodes changes as well. Such a behavior would make it unnecessarily hard to keep track of the evolution of different tree nodes. Thus, it is of utmost importance to ensure that our tree visualizations exhibit a stable layout across all points in time.

Static Position Animation One way to achieve a stable layout is the *static position animation* approach [LSP08, WL08]. In it, all visual elements stay in the same place throughout the whole evolution and just grow and shrink within a fixed area reserved for them. This area is calculated based on the maximum size that the element will reach at any point in time. A downside of this approach is that it wastes lots of space when an element is not at its maximum size (or worse, not shown at all). Also, it may work well for certain visualizations such as treemaps but not for most other tree visualizations. For example, applying this approach to an icicle would lead to empty spaces between the rectangles if they are not at their maximum size. As such a layout might rather distract than help users, we decided to implement a relaxed version of it that we call *relative position animation*.

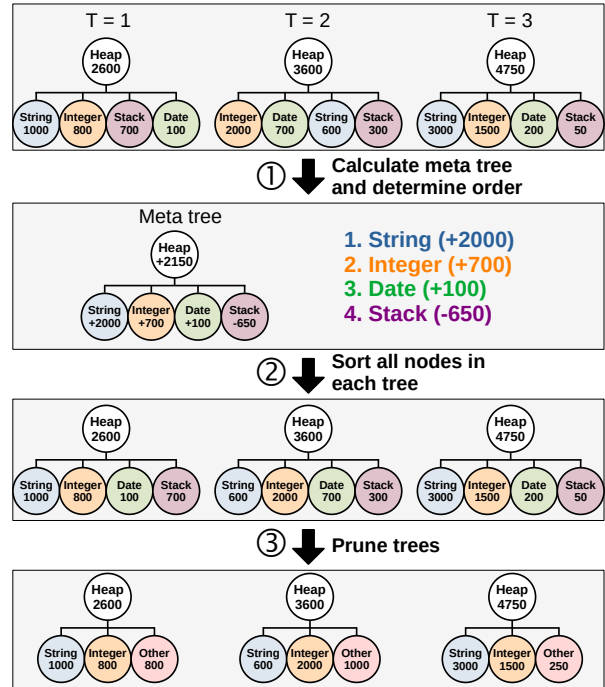


Figure 6: Preprocessing steps applied before heap evolution visualization: (1) meta tree calculation, (2) sorting, and (3) pruning.

Relative Position Animation Relative position animation means that the absolute position of a tree node might change when stepping through time, but the *order* of the nodes will stay the same. This is achieved by assigning a sort position (based on a certain criterion) to each node. This sort position is fixed across all points in time and is calculated for all nodes *once* after all memory trees have been computed. Weninger et al. [WGM19] already used a similar concept in another memory evolution visualization. There, they used various sorting strategies, including *start size* sorting, *end size* sorting and *absolute growth* sorting, which we also support for our tree visualizations. When applying the start size or the end size sorting strategy, all nodes in all trees are sorted by their object count or byte count at the start or end of the inspected time window respectively. Yet, we found that the *absolute growth* sorting strategy is usually the most useful strategy for investigating memory leaks. When applying the absolute growth sorting strategy, the absolute growth of each node between the first and the last point in time is calculated and stored in a meta tree. The first step in Figure 6 depicts how such a meta tree is calculated based on the nodes' absolute growth and how the meta tree is used to create a fixed sort order. In the second step of Figure 6, all nodes in each tree are sorted based on this sort order.

Tree Pruning Revisited In Section 4.1, we described tree pruning for a single memory tree by keeping the largest tree nodes. When pruning trees for heap evolution visualization, this pruning is slightly adjusted. Now, those nodes that have been ranked first *based on the sorting strategy* are preserved (instead of selecting them based on the current size). This ensures as few node additions and removals as possible between multiple points in time, while

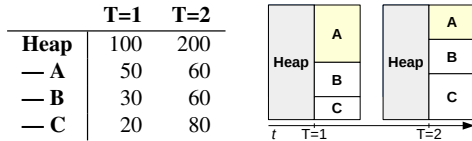


Figure 7: Unscaled visualizations may hide absolute growth.

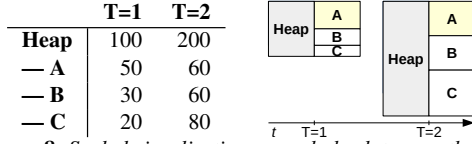


Figure 8: Scaled visualizations reveal absolute growth.

still benefiting from tree pruning. In the example in Figure 6, the sorted trees are pruned in the third step by preserving the first two nodes per tree, even though they might not be the biggest ones at that point in time (as is the case for *String* in the tree at $T = 2$).

5.2. Absolute Growth Visualization

A problem of our chosen tree visualizations is that they use the same amount of screen space to visualize any heap, independent of the heap’s size. Figure 7 illustrates this problem. At time $T = 1$, the heap has a size of 100MB, and at time $T = 2$ it has a size of 200MB, i.e., it doubled in size. Yet, the resulting visualizations do not reflect this. For example, by only looking at the tree visualizations, one may think that the number of A objects shrank between the two points in time, while the contrary is the case. Thus, we support to display *scaled* visualizations, as shown in Figure 8. For example, icicles are scaled along the y-axis based on the heap size at the respective point in time, where the largest heap within the analyzed time window uses the whole height. This should make it easier for users to comprehend the absolute growth of the heap over time.

Unscaled visualizations, i.e., visualizations that use the whole available space, are especially useful when text should be displayed. This is the reason why we use an unscaled visualization as the local view of our tool, which will be explained in more detail in Section 5.3. Scaled visualizations are more helpful when comparing the heap state of a system at multiple points in time, thus the timeline-based approach (which will be explained in more detail in Section 5.4) uses a scaled visualization by default.

5.3. Time-Travel-based Approach

In the context of visualization, Wettel and Lanza [WL08] define *time traveling* as stepping back and forth through time while the visualization updates itself to reflect the current state. Other memory evolution visualization approaches [WMM19, WMM20] also successfully used time travelling as their means of evolution visualization, which inspired us to also use this interaction technique for our memory tree evolution visualization.

Figure 9 shows a snapshot of our tool. On the bottom half the tool shows our time-travel-based visualization. It shows the heap at the currently selected point in time, once in (1) local mode and once in (2) global mode. Users are provided with (3) buttons to go to the next and the previous heap state, as well as a slider to

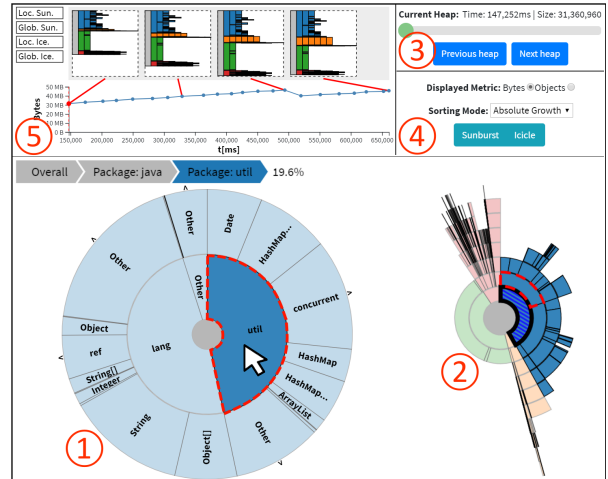


Figure 9: Overview of our visualization tool. The bottom-left visualization shows (1) the drilled-down local view of the heap, the bottom-right visualization shows (2) the global view with highlighted drill-down node and highlighted hover node. On the top right, the (3) time controls and the (4) visualization options can be found. Beside them, the (5) timeline-based visualization is situated.

move through time. When stepping through time, the visualization updates itself to show the memory tree at the given time. To make these updates more appealing, we leverage various animation features. For example, when the visualized data changes, the existing nodes do not snap to their new location but their positions and their sizes are gradually adjusted to match their new values, an approach called *tweening* [Wil09]. If a node was selected for drill-down, this selection is also preserved when stepping through time. At any time during analysis, the user (4) can switch between the different visualizations, i.e., sunburst and icicle (beside being able to change other settings). The visualizations are synchronized, i.e., the current drill-down node (if any) and all user-chosen settings such as metric (i.e., either object count or byte count) are preserved. Thus, users are able to continue at the exact same state of the analysis at which they were before they switched the visualization type.

5.4. Timeline-based approach

While the time-travel-based approach displays only one tree visualization (or more specifically, two, i.e., the local and the global view) at a time, the timeline-based visualization displays multiple tree visualizations, each representing the heap at a different point in time, in juxtaposition side-by-side. The name *timeline-based* stems from the fact that users can select which points in time to visualize by selecting them on a time-series chart, i.e., on a timeline. By comparing these tree visualizations with each other, the user should be able to detect changes over time in the heap composition.

In the upper left part of Figure 9 at (5), our timeline-based visualization can be seen. We display the overall heap consumption in a time-series line chart with clickable data points. Clicking them toggles the visualization of the tree visualization of the heap at the respective point in time. The tree at the currently selected point

in time is always shown in the timeline. Since the visualizations in the timeline view are quite small, it is not possible to display reasonable sized text within the node elements. Nevertheless, comparing the visualizations with each other quickly provides insight into the general evolution of the heap. Thus, the timeline-based visualization (overview) and the local view of the time-travel-based visualization (detailed analysis) complement each other well.

Figure 10 shows another example of the timeline view with four icicle visualizations, each depicting the heap state at a different point in time. The trees have been generated using the *type* classifier, followed by the *allocation site* classifier. This example uses *scaled* icicle visualizations, as explained in Section 5.2. Thus, the growth of the grey rectangle reflects the heap’s overall growth. As the first level represents objects of different types, we can see that at any point in time the heap mostly consists of four different types. Looking at the second level, we can see that the blue and the orange types have two different allocation sites each (where one of the two created far more objects than the other one), while the green and violet objects all have been allocated within a single method each.

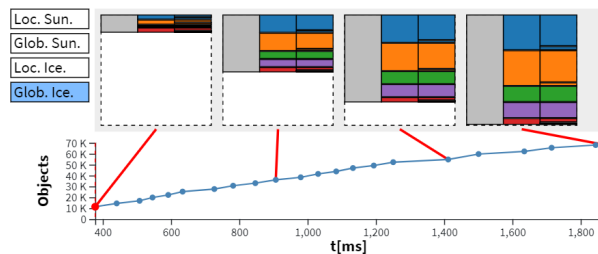


Figure 10: The timeline view shows the evolution of the heap over time by showing its state at multiple points in time in juxtaposition.

6. Implementation Details

We implemented our approach as a Javascript web application that heavily uses the *D3.js* library. This library provides many utility functions to create hierarchical visualizations [BOH11, CPRG16]. The implemented web application was integrated into AntTracks, which is a JavaFX-based application, using a JavaFX *WebView*. After loading a trace file, the user can select a time window and a list of classifiers which are used to classify the heap at multiple points within the selected time window. Subsequently, the resulting sequence of memory trees is converted to *JSON* and sent to the tree visualization web application via *WebSockets* [FM11, WPJR11]. Using this *JSON* interface, our visualization tool could also be used by other monitoring tools than AntTracks. A prototype of the tool can be found at <http://bit.ly/STAG-MemoryTreeVizTool>. This prototype also contains the data used in the following two case studies. A video explaining the tool can be found at <http://bit.ly/STAG-MemoryTreeVizVideo>.

7. Case Studies

To demonstrate the feasibility of our approach, we searched online for real-world applications that contain memory leaks to showcase how to investigate them. In the following, we present the analysis of a memory leak in the *Commons HttpClient* library, as well as the analysis of a memory leak in the *Dynatrace easyTravel* application.

7.1. Commons HttpClient

Finding applications or libraries that contain memory leaks requires lots of effort, since their source code and the needed build tools have to be publicly available. To find the memory leaking library discussed in this section, we browsed Apache’s issue tracker [Apa20] for the keyword *leak*. This way, we found an old issue regarding a memory leak in the *Commons HttpClient* library, a library that can be used to send HTTP requests. As the library was completely unknown to us authors, it seemed like a good example to check if our tree visualizations are useful to detect accumulating objects even in an unknown application. We downloaded the affected version 3.0.1 [Apa06] and built a small driver application [WM20], which creates HTTP connections in multiple batches, where in each batch 10,000 connections are created and deleted shortly thereafter.

To analyze the memory behavior of our driver application, we recorded a memory trace and inspected it using AntTracks. One would expect to see spikes in the memory usage, as it should go up when the connections are created and should go down after their deletion. Yet, contrary to this expectation, Figure 11 shows that the memory consumption continuously rose. It seems as if only a part of the objects that were allocated during every batch are actually garbage-collected afterwards. To create the tree visualizations in Figure 12 through Figure 14, we grouped the heap objects by *type* and *allocation site* and sorted them by *absolute growth*.

We started our analysis by comparing the first two sunbursts in Figure 12. The first sunburst depicts the heap at the first memory consumption peak, while the second sunburst depicts the heap at the first dip. What immediately catches one’s eye is that the percentage of the heap that is occupied by the red type (i.e., the memory tree’s artificial *Other* node) shrank significantly between the peak and the dip, while the relative amount of memory occupied by objects of the other types grew. Looking at the next two sunbursts we can see that this trend continues. In the end, around 90% of the heap are occupied by objects of only six different types. The local view of the time-travel-based visualizations at this point in time is also shown in Figure 13. All of these types except *HostParams* (brown) are allocated at a single allocation site each (since all types only have a single node on the second level).

To better grasp how the absolute sizes of the nodes develop over time, the *scaling feature* of our icicle views is used in Figure 14. Comparing the first icicle to the last icicle immediately indicates that the heap grew several-fold. Furthermore, we again can see that nearly the complete growth accounts to six different types.

To find the reason for the memory leak, the allocation sites of the types that grew the most are inspected. We can see that nearly all *LinkedList* objects were allocated in the constructor of *MultiThreadedHttpClientConnectionManager\$HostConnectionPool*, which is the type that grew the third most (green). The main allocation site of that type is the method *MultiThreadedHttpClientConnectionManager\$ConnectionPool::getHostPool()*. This provided us with enough information to investigate that method in the source code. There we found that the *MultiThreadedHttpClientConnectionManager\$HostConnectionPool* objects are added to a map, yet they are not removed from that map when the connection is

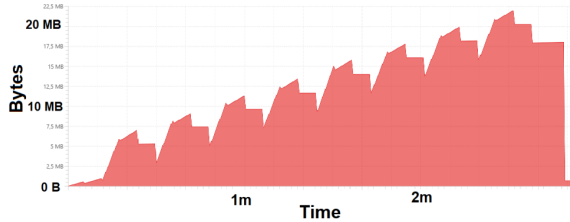


Figure 11: AntTracks reports continuous memory growth in HttpClient.

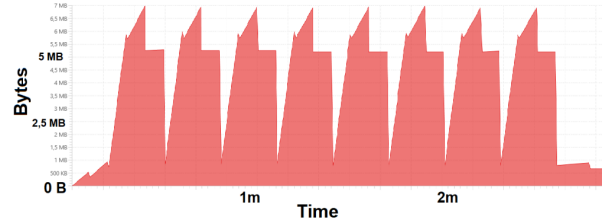


Figure 15: Expected spike pattern after fixing the memory leak.

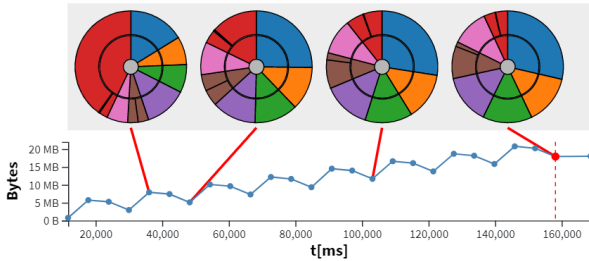


Figure 12: Timeline view with unscaled sunbursts.

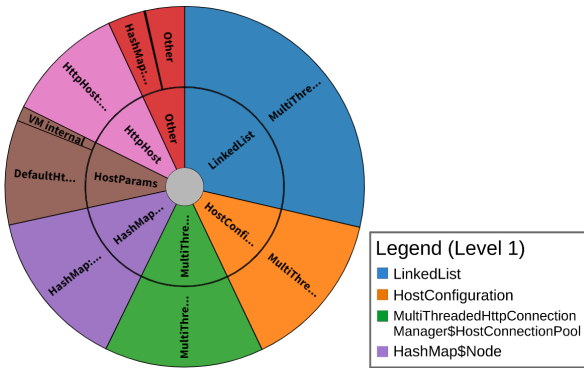


Figure 13: Final sunburst at the end of the time window.

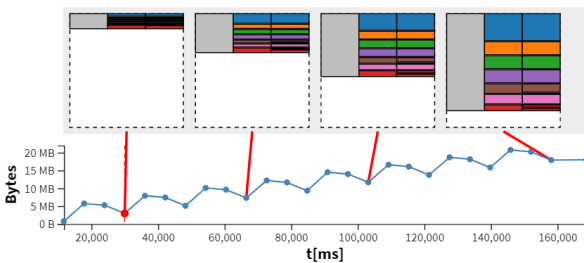


Figure 14: Timeline view with scaled icicles.

deleted, resulting in a memory leak. This causes the connection pool (and all other objects referenced by it) to accumulate over time. Fixing the code by correctly removing the objects from the map once the connections are closed gets rid of the memory leak and leads to the expected memory behavior, as shown in Figure 15.

7.2. easyTravel

The second investigated application is Dynatrace easyTravel. Dynatrace focuses on application performance monitoring (APM) and

distributes easyTravel as their state-of-the-art demo application. It is a multi-tier application for a travel agency, using a Java back-end. An automatic load generator distributed together with easyTravel can simulate accesses to the service. When easyTravel is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

Our heap evolution visualization grouped the heap at multiple points in time using three classifiers: *containing data structure*, *type*, and *closest domain call site*. The first classifier groups objects based on the data structure(s) they are contained in. If an object is contained in a single data structure it is assigned the group “<Data structure type> (allocated in <Data structure allocation site>)”, for example “HashMap (allocated in MyClass::myMethod())”, otherwise it is either assigned the group “Not contained in a data structure” or “Contained in multiple data structures”. This way, single data structures that keep alive a lot of objects can easily be detected. The third classifier, i.e., *closest domain call site*, differs from the normal allocation site classifier as it returns the method call within easyTravel’s code base that caused the allocation even if the allocation itself is hidden inside a third-party framework.

Figure 16 shows the local view of our time-travel-based sunburst visualization at three different points in time. The nodes within the trees have been sorted by *absolute growth*, i.e., independent of which sunburst we look at, we can automatically infer that the object group represented by the blue segment (i.e., objects stored in a ConcurrentHashMap data structure that has been allocated in method findLocations() of class JourneyService) grows the most over the selected time window. This also becomes apparent when comparing the sunburst at time t_1 to the sunburst at time t_3 . While all other data structure segments (inner circle segments) shrink, the segment of the suspicious ConcurrentHashMap grows strongly. By hovering over the circle segment at both points in time, we can find out that the ConcurrentHashMap only makes up 11.8% of the heap at t_1 , while it makes up 39.5% at t_3 .

Figure 17 shows the sunburst at t_3 , with a drill-down performed on the ConcurrentHashMap. We can see that around 45% of the objects stored in this data structure are each of type Location and Date. Due to the drill-down, we can now also see the third tree level, i.e., the *closest domain call sites*. Since Location as well as Date both only have one child node, we know that all allocations of these objects are caused by a single method each.

The collected information greatly helps to locate and fix the problem. The method in which the Location objects depicted in Figure 17 are added to the suspicious ConcurrentHashMap was easily found. Its name locationCache and its use in the code

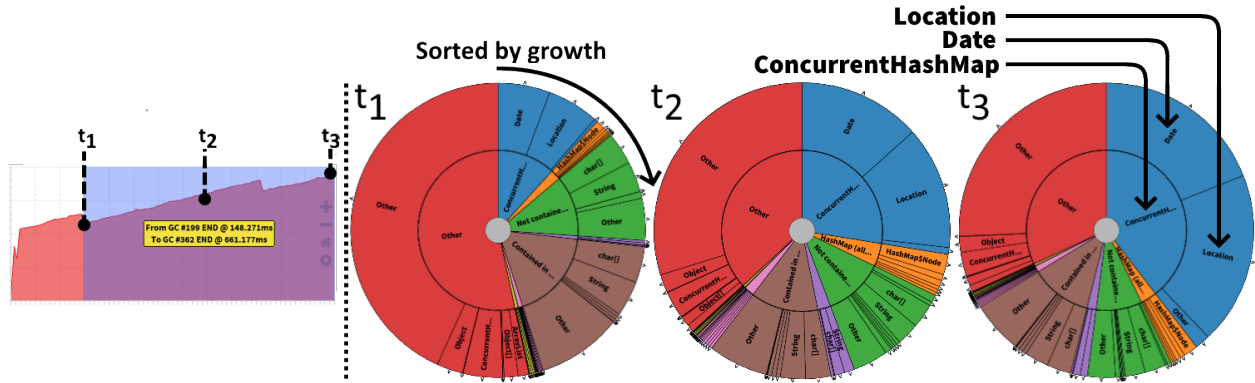


Figure 16: Heap evolution time travel through easyTravel shown at three different points in time t_1 , t_2 , and t_3 . This indicates a leak involving a data structure of type `ConcurrentHashMap` that has been allocated in method `findLocations()` of class `JourneyService` (inner blue circle segment). This data structure accumulates `Location` and `Date` objects over time (outer blue circle segments).

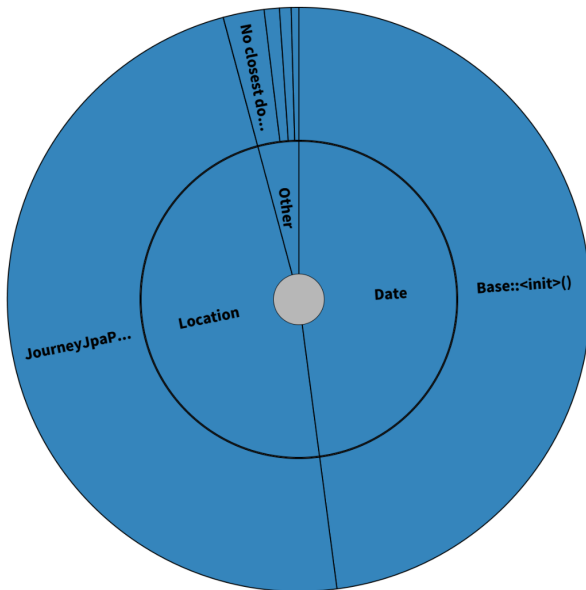


Figure 17: Drill-down into the suspicious `ConcurrentHashMap` data structure at timestamp t_3 .

reveal that this map should serve as a cache for location searches. Once a search has been executed for a given `QueryKey`, a list of `Location` objects is stored. Subsequent searches for the same key should find the respective entry in the map. However, `QueryKey` neither implements `hashCode` nor `equals`. Thus, every request (even for an already existing key) resulted in a cache miss, which led to this typical memory leak. We were able to easily resolve this problem by implementing the two mentioned methods accordingly.

8. Related Work

As ample work regarding tree visualizations has already been presented throughout this work, this section focuses on visualizations in the domain of memory monitoring. Most memory visualizations

revolve around object (reference) graph visualization. A pure object graph consists of nodes representing heap objects and edges that represent the references between them [PNC98]. Even though such a graph could be directly visualized as a node-link diagram [ZZ01], the size of modern applications (having millions of live objects) renders approaches that display every heap object as a separate node infeasible. Thus, most approaches create *ownership trees* using the concept of *object ownership* [PNC98, Mit06, WGM18b] based on the *dominator relation* [LT79]. Ownership trees can be used to detect objects that keep many other objects alive.

Reiss [Rei09, Rei10] visualizes the aggregated ownership graph in an icicle-like visualization using coloring, hatching, hue and saturation. The approach by Hill et al. [HNP00, HNP02] plots the evolution of ownership trees in a scalable tree visualization that shares visual similarities with flame graph [Gre16a, Gre16b]. Mitchell et al. [MSS09] apply further transformations on ownership trees to detect costly data structures, which are then displayed in a node-link diagram. Heapviz [AKG*10, KAG*13] is a tool that also displays data structures on different levels of detail, arranging collapsible nodes in a radial node-link diagram. The work by De Pauw and Sevitsky [DPS00] is one of the few object graph visualization approaches that does not utilize the dominator relation. Instead, they extract reference patterns, i.e., repetitive reference sequences in the heap object graph, and visualize occurrences of these patterns. The detection of these patterns can be restricted to those objects that have been created between two heap snapshots (i.e., potentially leaking objects), which then can be explored visually.

Our approach is orthogonal to these existing visualization approaches. Most object reference graph visualizations focus on the analysis of the keep-alive relation between objects (e.g., which objects keep alive objects of type B). Yet, this expects the user to already know which objects need to be inspected in more detail. This is where our approach comes into play. It gives the user visual information about which objects accumulate over time, as these are the objects that are most likely the result of a memory leak. For example, it may report that objects of type B that are allocated in method `MyClass.myMethod()` consistently increased in num-

ber over time. Thus, the main focus of this work lies in the *detection* of growing heap object groups. Nevertheless, the information provided by our visualizations may not only help in detecting growing object groups, but are also often already detailed enough to help in *fixing* a leak. As shown in Section 7, knowing which kinds of objects accumulate, where these accumulating objects are allocated and in which data structures they are stored (information that can be provided by our approach) is often enough to be able to locate and fix a problem in the source code. Thus, our approach can be used on its own, or in combination with existing graph-based analysis techniques if additional information is needed or wanted.

9. Current Limitations and Future Work

In this section, we discuss current limitations of our work and our tool and how we will address them in the future.

9.1. User Study

We believe that the presented techniques are useful to inspect memory evolution over time, especially for novice users that could otherwise easily be overwhelmed if the visualized data was presented in raw format or tables. We presented case studies to demonstrate how tree visualization can be used in the domain of memory monitoring and how users are able to understand and reason about the memory behavior of real world applications. Even though existing work suggests that tree visualizations are useful for a variety of analysis tasks [WTM06, Teo07, BPP17], a more thorough evaluation is planned. We want to conduct a user study to compare the performance of participants who use tree visualizations to inspect an application's memory behavior with the performance of participants who use other graphical and textual representations.

9.2. Information Highlighting and Guidance

Even though information visualization has the potential to ease the analysis of the underlying data, a person still requires a fair amount of background knowledge and experience to perform memory analysis effectively. Especially novice users often lack this experience and consequently struggle when using memory analysis tools [WGGSS20]. In the future, we want to further increase the accessibility of our tree visualizations by making their use easier. The tool should automatically detect suspicious memory behavior, e.g., growing object groups, and then *guide* the user through the analysis by *explaining* the steps that have to be performed, alongside automatic *highlighting* of important information in the visualization.

9.3. Reference Visualization

As shown, our current visualizations are a great way to detect and inspect growth over time. Yet, once we know which objects accumulate over time (often objects of a few different types allocated at a few different allocation sites), most of the further analysis happens on the source code level in the IDE. Objects accumulate over time if they are directly or indirectly referenced by a GC root. For example, as presented in Section 7.1, objects of one type that were stored in a map caused objects of multiple different types to accumulate. Thus, the references between objects can be a vital information in the analysis of memory leaks. In the future, we plan to

extend our tree visualization to support the depiction of references between object groups, for example by using tree visualization that support hierarchical edge bundling [Hol06, HCvW07, HdRFH12].

9.4. Data Structure Growth Visualization

Currently, our visualization tool can display the evolution of the whole heap over time by grouping the live heap objects based on a list of user-defined classifiers at multiple points in time. In the future, we want to explore how to use the same tree visualization techniques to display the results of existing analysis features that yet lack visualization support. Inspired by related work on data structure visualization [AKG*10, KAG*13], one of AntTracks's analysis features that we want to enrich using our visualizations is its *automatic data structure growth analysis* [WGM18a, WGM19a]. This feature automatically detects strongly growing data structures in a monitored application and reports them to the user for more detailed inspection using drill-down operations within a tree table. Since the visualization of evolutionary data as well as drilling down on the data are core features of our approach, we plan to integrate our tree visualizations with the existing data structure analysis.

10. Conclusions

In this paper, we presented our approach to apply *tree visualizations* to facilitate the analysis of *memory leaks*. We discussed how a heap state, more specifically its heap objects, can be grouped into a memory tree, and how such a tree can be visualized using existing tree visualization techniques. We defined a requirements catalog that we used to select the *sunburst* and the *icicle* visualization techniques as suitable to display heap memory. We then presented techniques how to reduce a memory tree's complexity by pruning it and how a drill-down functionality can be used in the selected visualizations to enable detailed analyses of the heap composition. Our approach is not only able to display a single heap state, but can also visualize the memory evolution over time in two different ways: a timeline-based approach that displays the visualized state of the heap at multiple point in time side-by-side, and a time-travel-based approach for detailed analyses. Growing elements in these visualizations hint at an accumulation of heap objects that could be the result of a possible memory leak.

We implemented our approach as a D3.js web application that supports various convenience features such as animations when moving between points in time. We presented case studies in which we performed memory analyses of different applications to show the approach's feasibility and usefulness. We hope that our tree visualizations can aid experienced users as well as users with a limited background in memory analysis in visually inspecting and analyzing the memory behavior of their applications.

Acknowledgment

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

- [AHL*13] AUBER D., HUET C., LAMBERT A., RENOUST B., SAL-LABERRY A., SAULNIER A.: GosperMap: Using a Gosper Curve for Laying Out Hierarchical Data. *IEEE Trans. Vis. Comput. Graph.* 19, 11 (2013), 1820–1832. URL: <https://doi.org/10.1109/TVCG.2013.91>, doi:10.1109/TVCG.2013.91.3
- [AKG*10] AFTANDILIAN E., KELLEY S., GRAMAZIO C., RICCI N. P., SU S. L., GUYER S. Z.: Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the ACM Symposium on Software Visualization (SOFTVIS)* (2010), pp. 53–62. URL: <https://doi.org/10.1145/1879211.1879222>, doi:10.1145/1879211.1879222.9,10
- [Apa06] APACHE SOFTWARE FOUNDATION: Commons HttpClient version 3.0.1, 2006. URL: <https://mvnrepository.com/artifact/commons-httpclient/commons-httpclient/3.0.1.7>
- [Apa20] APACHE SOFTWARE FOUNDATION: Issue tracker for HttpClient, 2020. URL: <https://issues.apache.org/jira/projects/HTTPCLIENT/issues.7>
- [BAB18] BLANCO A. F., ALCOCER J. P. S., BERGEL A.: Effective Visualization of Object Allocation Sites. In *Proc. of the IEEE Working Conference on Software Visualization (VISSOFT)* (2018), pp. 43–53. URL: <https://doi.org/10.1109/VISSOFT.2018.00013>, doi:10.1109/VISSOFT.2018.00013.1
- [BLM15] BITTO V., LENGAUER P., MÖSSENBOECK H.: Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Int'l. Conf. on Principles and Practices of Programming on The Java Platform (PPPJ)* (2015), pp. 76–89. URL: <https://doi.org/10.1145/2807426.2807433>, doi:10.1145/2807426.2807433.2
- [BN01] BARLOW S. T., NEVILLE P.: A Comparison of 2-D Visualizations of Hierarchies. In *Proc. of the IEEE Symposium on Information Visualization (INFOVIS)* (2001), pp. 131–138. URL: <https://doi.org/10.1109/INFVIS.2001.963290>, doi:10.1109/INFVIS.2001.963290.3
- [BNK16] BACHER I., NAMEE B. M., KELLEHER J. D.: Using Icicle Trees to Encode the Hierarchical Structure of Source Code. In *Proc. of the Eurographics Conf. on Visualization (EuroVis)* (2016), pp. 97–101. URL: <https://doi.org/10.2312/eurovisshort.20161168>, doi:10.2312/eurovisshort.20161168.4
- [BOH11] BOSTOCK M., OGIEVETSKY V., HEER J.: D³ Data-Driven Documents. *IEEE Trans. Vis. Comput. Graph.* 17, 12 (2011), 2301–2309. URL: <https://doi.org/10.1109/TVCG.2011.185>, doi:10.1109/TVCG.2011.185.7
- [BPP17] BIUK-AGHAI R. P., PANG P. C., PANG B.: Map-like Visualisations vs. Treemaps: An Experimental Comparison. In *Proc. of the 10th Int'l. Symposium on Visual Information Communication and Interaction (VINCI)* (2017), ACM, pp. 113–120. URL: <https://doi.org/10.1145/3105971.3105976>, doi:10.1145/3105971.3105976.10
- [Bru09] BRUTLAG J.: Speed Matters for Google Web Search, 2009. URL: <https://ai.googleblog.com/2009/06/speed-matters.html.3>
- [CM07] CAWTHON N., MOERE A. V.: The Effect of Aesthetic on the Usability of Data Visualization. In *Proc. of the 11th Int'l. Conf. on Information Visualisation (IV)* (2007), pp. 637–648. URL: <https://doi.org/10.1109/IV.2007.147>, doi:10.1109/IV.2007.147.3
- [CPRG16] CALLEYA J., PAWLING R., RYAN C., GASPAR H. M.: Using Data Driven Documents (D3) to Explore a Whole Ship Model. In *Proc. of the 11th System of Systems Engineering Conf. (SoSE)* (2016), pp. 1–6. URL: <https://doi.org/10.1109/SYSE.2016.7542947>, doi:10.1109/SYSE.2016.7542947.7
- [CZvDR09] CORNELISSEN B., ZAIDMAN A., VAN DEURSEN A., ROMPAEY B. V.: Trace Visualization for Program Comprehension: A Controlled Experiment. In *Proc. of the 17th IEEE Int'l. Conf. on Program Comprehension (ICPC)* (2009), pp. 100–109. URL: <https://doi.org/10.1109/ICPC.2009.5090033>, doi:10.1109/ICPC.2009.5090033.1
- [DPS00] DE PAUW W., SEVITSKY G.: Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency - Practice and Experience* 12, 14 (2000), 1431–1454. URL: [https://doi.org/10.1002/1096-9128\(20001210\)12:14<1431::AID-CPE542>3.0.CO;2-2](https://doi.org/10.1002/1096-9128(20001210)12:14<1431::AID-CPE542>3.0.CO;2-2), doi:10.1002/1096-9128(20001210)12:14<1431::AID-CPE542>3.0.CO;2-2.9
- [Ecl20] ECLIPSE FOUNDATION: Eclipse Memory Analyzer (MAT), 2020. URL: <https://www.eclipse.org/mat/.1>
- [FFHW15] FITTKAU F., FINKE S., HASSELBRING W., WALLER J.: Comparing Trace Visualizations for Program Comprehension Through Controlled Experiments. In *Proc. of the 23rd IEEE Int'l. Conf. on Program Comprehension (ICPC)* (2015), pp. 266–276. URL: <https://doi.org/10.1109/ICPC.2015.37>, doi:10.1109/ICPC.2015.37.1
- [FKH15] FITTKAU F., KRAUSE A., HASSELBRING W.: Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment. In *Proc. of the 3rd IEEE Working Conf. on Software Visualization (VISSOFT)* (2015), pp. 36–45. URL: <https://doi.org/10.1109/VISSOFT.2015.7332413>, doi:10.1109/VISSOFT.2015.7332413.1
- [FKH17] FITTKAU F., KRAUSE A., HASSELBRING W.: Software Landscape and Application Visualization for System Comprehension with ExplorViz. *Inf. Softw. Technol.* 87 (2017), 259–277. URL: <https://doi.org/10.1016/j.infsof.2016.07.004>, doi:10.1016/j.infsof.2016.07.004.1,3
- [FM11] FETTE I., MELNIKOV A.: The WebSocket Protocol. *RFC 6455* (2011), 1–71. URL: <https://doi.org/10.17487/RFC6455>, doi:10.17487/RFC6455.7
- [GCS*20] GHANAVATI M., COSTA D., SEBOEK J., LO D., ANDRZEJAK A.: Memory and Resource Leak Defects and their Repairs in Java Projects. *Empirical Software Engineering* 25, 1 (2020), 678–718. URL: <https://doi.org/10.1007/s10664-019-09731-8>, doi:10.1007/s10664-019-09731-8.1
- [Gre16a] GREGG B.: The Flame Graph. *ACM Queue* 14, 2 (2016), 10. URL: <https://doi.org/10.1145/2927299.2927301>, doi:10.1145/2927299.2927301.9
- [Gre16b] GREGG B.: The Flame Graph. *Commun. ACM* 59, 6 (2016), 48–57. URL: <https://doi.org/10.1145/2909476>, doi:10.1145/2909476.9
- [HBO10] HEER J., BOSTOCK M., OGIEVETSKY V.: A Tour through the Visualization Zoo. *ACM Queue* 8, 5 (2010), 20. URL: <http://doi.acm.org/10.1145/1794514.1805128>, doi:10.1145/1794514.1805128.2,4
- [HCvW07] HOLTEN D., CORNELISSEN B., VAN WIJK J. J.: Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views. In *Proc. of the 4th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2007), pp. 47–54. URL: <https://doi.org/10.1109/VISSOFT.2007.4290699>, doi:10.1109/VISSOFT.2007.4290699.10
- [HdRFH12] HOP W., DE RIDDER S., FRASINCAR F., HOGENBOOM F.: Using Hierarchical Edge Bundles to Visualize Complex Ontologies in GLOW. In *Proc. of the ACM Symposium on Applied Computing (SAC)* (2012), pp. 304–311. URL: <https://doi.org/10.1145/2245276.2245338>, doi:10.1145/2245276.2245338.10
- [HHK*17] HINIKER A., HONG S. R., KIM Y., CHEN N., WEST J. D., ARAGON C. R.: Toward the Operationalization of Visual Metaphor. *J. Assoc. Inf. Sci. Technol.* 68, 10 (2017), 2338–2349. URL: <https://doi.org/10.1002/asi.23857>, doi:10.1002/asi.23857.3
- [HNP00] HILL T., NOBLE J., POTTER J.: Scalable Visualisations

- with Ownership Trees. In *Proc. of the 37th Int'l. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS)* (2000), pp. 202–213. URL: <https://doi.org/10.1109/TOOLS.2000.891370>, doi:10.1109/TOOLS.2000.891370. 9
- [HNPO2] HILL T., NOBLE J., POTTER J.: Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.* 13, 3 (2002), 319–339. URL: <https://doi.org/10.1006/jvlc.2002.0238>, doi:10.1006/jvlc.2002.0238. 9
- [Hol06] HOLTEN D.: Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Trans. Vis. Comput. Graph.* 12, 5 (2006), 741–748. URL: <https://doi.org/10.1109/TVCG.2006.147>, doi:10.1109/TVCG.2006.147. 10
- [HTMD14] HAHN S., TRÜMPER J., MORITZ D., DÖLLNER J.: Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps. In *Proc. of the 5th Int'l. Conf. on Information Visualization Theory and Applications (IVAPP)* (2014), pp. 50–58. URL: <https://doi.org/10.5220/0004686200500058>, doi:10.5220/0004686200500058. 3
- [JS91] JOHNSON B., SHNEIDERMAN B.: Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Proc. of the IEEE Conf. on Visualization* (1991), pp. 284–291. URL: <https://doi.org/10.1109/VISUAL.1991.175815>, doi:10.1109/VISUAL.1991.175815. 2
- [KAG*13] KELLEY S., AFTANDILIAN E., GRAMAZIO C., RICCI N. P., SU S. L., GUYER S. Z.: Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. *Information Visualization* 12, 2 (2013), 163–177. URL: <https://doi.org/10.1177/1473871612438786>, doi:10.1177/1473871612438786. 9, 10
- [Kit94] KITCHIN R. M.: Cognitive maps: What are they and why study them? *Journal of Environmental Psychology* 14, 1 (1994), 1–19. URL: [https://doi.org/10.1016/S0272-4944\(05\)80194-X](https://doi.org/10.1016/S0272-4944(05)80194-X), doi:10.1016/S0272-4944(05)80194-X. 3
- [KK91] KAMADA T., KAWAI S.: A General Framework for Visualizing Abstract Objects and Relations. *ACM Trans. Graph.* 10, 1 (1991), 1–39. URL: <https://doi.org/10.1145/99902.99903>, doi:10.1145/99902.99903. 2
- [Lak94] LAKOFF G.: *Master Metaphor List*. University of California, 1994. 3
- [LBF*16] LENGAUER P., BITTO V., FITZEK S., WENINGER M., MÖSSENBOECK H.: Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)* (2016), pp. 4:1–4:11. URL: <https://doi.org/10.1145/2972206.2972220>, doi:10.1145/2972206.2972220. 2
- [LBM15] LENGAUER P., BITTO V., MÖSSENBOECK H.: Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2015), pp. 51–62. URL: <https://doi.org/10.1145/2668930.2688037>, doi:10.1145/2668930.2688037. 2
- [LBM16] LENGAUER P., BITTO V., MÖSSENBOECK H.: Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2016), pp. 249–260. URL: <https://doi.org/10.1145/2851553.2851555>, doi:10.1145/2851553.2851555. 2
- [LH14] LIU Z., HEER J.: The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.* 20, 12 (2014), 2122–2131. URL: <https://doi.org/10.1109/TVCG.2014.2346452>, doi:10.1109/TVCG.2014.2346452. 3
- [LSDT19] LIMBERGER D., SCHEIBEL W., DÖLLNER J., TRAPP M.: Advanced Visual Metaphors and Techniques for Software Maps. In *Proc. of the 12th Int'l. Symposium on Visual Information Communication and Interaction (VINCI)* (2019), pp. 11:1–11:8. URL: <https://doi.org/10.1145/3356422.3356444>, doi:10.1145/3356422.3356444. 3
- [LSP08] LANGELIER G., SAHRAOUI H. A., POULIN P.: Exploring the Evolution of Software Quality with Animated Visualization. In *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC)* (2008), pp. 13–20. URL: <https://doi.org/10.1109/VLHCC.2008.4639052>, doi:10.1109/VLHCC.2008.4639052. 5
- [LT79] LENGAUER T., TARJAN R. E.: A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141. URL: <https://doi.org/10.1145/357062.357071>, doi:10.1145/357062.357071. 9
- [Mit06] MITCHELL N.: The Runtime Structure of Object Ownership. In *Proc. of the 20th European Conf. on Object-oriented Programming (ECOOP)* (2006), pp. 74–98. URL: https://doi.org/10.1007/11785477_5, doi:10.1007/11785477_5. 9
- [MSS09] MITCHELL N., SCHONBERG E., SEVITSKY G.: Making Sense of Large Heaps. In *Proc. of the 23rd European Conf. on Object-Oriented Programming (ECOOP)* (2009), pp. 77–97. URL: https://doi.org/10.1007/978-3-642-03013-0_5, doi:10.1007/978-3-642-03013-0_5. 9
- [Mur13] MURRAY S.: *Interactive Data Visualization for the Web*. O'Reilly Media, Inc., 2013. 2
- [Ora20] ORACLE: VisualVM: All-in-One Java Troubleshooting Tool, 2020. URL: <https://visualvm.github.io/>. 1
- [PNC98] POTTER J., NOBLE J., CLARKE D. G.: The Ins and Outs of Objects. In *Proc. of the Australian Software Engineering Conf. (ASWEC)* (1998), pp. 80–89. URL: <https://doi.org/10.1109/ASWEC.1998.730915>, doi:10.1109/ASWEC.1998.730915. 9
- [Rei09] REISS S. P.: Visualizing the Java Heap to Detect Memory Problems. In *Proc. of the 5th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2009), pp. 73–80. URL: <https://doi.org/10.1109/VISSOFT.2009.5336418>, doi:10.1109/VISSOFT.2009.5336418. 9
- [Rei10] REISS S. P.: Visualizing the Java Heap. In *Proc. of the 32nd ACM/IEEE Int'l. Conf. on Software Engineering* (2010), pp. 251–254. URL: <https://doi.org/10.1145/1810295.1810344>, doi:10.1145/1810295.1810344. 9
- [Sch11] SCHULZ H.: Treevis.net: A Tree Visualization Reference. *IEEE Computer Graphics and Applications* 31, 6 (2011), 11–15. URL: <https://doi.org/10.1109/MCG.2011.103>, doi:10.1109/MCG.2011.103. 2
- [SS06] SCHULZ H., SCHUMANN H.: Visualizing Graphs - A Generalized View. In *Proc. of the 10th Int'l. Conf. on Information Visualization (IV)* (2006), IEEE Computer Society, pp. 166–173. URL: <https://doi.org/10.1109/IV.2006.130>, doi:10.1109/IV.2006.130. 2
- [STLD20] SCHEIBEL W., TRAPP M., LIMBERGER D., DÖLLNER J.: A Taxonomy of Treemap Visualization Techniques. In *Proc. of the 15th Int'l. Joint Conf. on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)* (2020), pp. 273–280. URL: <https://doi.org/10.5220/0009153902730280>, doi:10.5220/0009153902730280. 2
- [SZ00] STASKO J. T., ZHANG E.: Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations. In *Proc. of the IEEE Symposium on Information Visualization (INFOVIS)* (2000), pp. 57–65. URL: <https://doi.org/10.1109/INFVIS.2000.885091>, doi:10.1109/INFVIS.2000.885091. 2, 3
- [Teo07] TEOH S. T.: A Study on Multiple Views for Tree Visualization. In *Proc. of SPIE - Visualization and Data Analysis* (2007), vol. 6495. URL: <https://doi.org/10.1117/12.703076>, doi:10.1117/12.703076. 10
- [TM04] TORY M., MÖLLER T.: Human Factors in Visualization Research. *IEEE Trans. Vis. Comput. Graph.* 10, 1 (2004), 72–84. URL: <https://doi.org/10.1109/TVCG.2004.1260759>, doi:10.1109/TVCG.2004.1260759. 4

- [War13] WARE C.: Chapter One - Foundations for an Applied Science of Data Visualization. In *Information Visualization (Third Edition)*, third edition ed., Interactive Technologies. Morgan Kaufmann, Boston, 2013, pp. 1 – 30. URL: <http://www.sciencedirect.com/science/article/pii/B9780123814647000016>, doi:<https://doi.org/10.1016/B978-0-12-381464-7.00001-6>. 2
- [Wen20] WENINGER M.: AntTracks, 2020. URL: <http://mevss.jku.at/AntTracks.2>
- [WGS20] WENINGER M., GRÜNBACHER P., GANDER E., SCHÖRGENHUMER A.: Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proc. ACM Hum.-Comput. Interact.* 4, EICS (June 2020). URL: <https://doi.org/10.1145/3394977>, doi:10.1145/3394977. 10
- [WKG10] WARD M. O., GRINSTEIN G. G., KEIM D. A.: *Interactive Data Visualization - Foundations, Techniques, and Applications*. A K Peters, 2010. URL: <http://www.akpeters.com/product.asp?ProdCode=4735.2>
- [WGM18a] WENINGER M., GANDER E., MÖSSENBOECK H.: Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symp. on Software Performance (SSP)* (2018), pp. 64–66. URL: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf.1,10
- [WGM18b] WENINGER M., GANDER E., MÖSSENBOECK H.: Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l. Conf. on Managed Languages & Runtimes (ManLang)* (2018), pp. 14:1–14:13. URL: <https://doi.org/10.1145/3237009.3237023>, doi:10.1145/3237009.3237023. 9
- [WGM19a] WENINGER M., GANDER E., MÖSSENBOECK H.: Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proc. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2019), pp. 273–284. URL: <https://doi.org/10.1145/3297663.3310297>, doi:10.1145/3297663.3310297. 1, 2, 10
- [WGM19b] WENINGER M., GANDER E., MÖSSENBOECK H.: Detection of Suspicious Time Windows In Memory Monitoring. In *Proc. of the 16th ACM SIGPLAN Int'l. Conf. on Managed Programming Languages and Runtimes (MPLR)* (2019), pp. 95–104. URL: <https://doi.org/10.1145/3357390.3361025>, doi:10.1145/3357390.3361025. 5
- [Wil09] WILLIAMS R.: *The Animator's Survival Kit—Revised Edition: A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Internet Animators*. Faber & Faber, Inc., 2009. 6
- [WL07] WETTEL R., LANZA M.: Visualizing Software Systems as Cities. In *Proc. of the 4th IEEE Int'l. Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)* (2007), IEEE Computer Society, pp. 92–99. URL: <https://doi.org/10.1109/VISSOFT.2007.4290706>, doi:10.1109/VISSOFT.2007.4290706. 3
- [WL08] WETTEL R., LANZA M.: Visual Exploration of Large-Scale System Evolution. In *Proc. of the 15th Working Conf. on Reverse Engineering (WCRE)* (2008), pp. 219–228. URL: <https://doi.org/10.1109/WCRE.2008.55>, doi:10.1109/WCRE.2008.55. 5, 6
- [WLM17] WENINGER M., LENGAUER P., MÖSSENBOECK H.: User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l. Conf. on Performance Engineering (ICPE)* (2017), pp. 357–360. URL: <https://doi.org/10.1145/3030207.3030236>, doi:10.1145/3030207.3030236. 1, 2
- [WLR11] WETTEL R., LANZA M., ROBBES R.: Software Systems as Cities: A Controlled Experiment. In *Proc. of the 33rd Int'l. Conf. on Software Engineering (ICSE)* (2011), ACM, pp. 551–560. URL: <https://doi.org/10.1145/1985793.1985868>, doi:10.1145/1985793.1985868. 1, 3
- [WM18] WENINGER M., MÖSSENBOECK H.: User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 2018 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2018), pp. 115–126. URL: <https://doi.org/10.1145/3184407.3184412>, doi:10.1145/3184407.3184412. 1, 2
- [WM20] WENINGER M., MAKOR L.: HttpClient Leak Driver, 2020. URL: <https://github.com/NeonMika/httpclient-leak-driver/>. 7
- [WMM19] WENINGER M., MAKOR L., GANDER E., MÖSSENBOECK H.: AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Comp. of the 2019 ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE)* (2019), pp. 29–32. URL: <https://doi.org/10.1145/3302541.3313100>, doi:10.1145/3302541.3313100. 1, 5
- [WMM20] WENINGER M., MAKOR L., MÖSSENBOECK H.: Memory Leak Visualization using Evolving Software Cities. In *Proc. of the 10th Symp. on Software Performance (SSP)* (2019), pp. 44–46. URL: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf.3,6
- [WMM20] WENINGER M., MAKOR L., MÖSSENBOECK H.: Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proc. of the 8th IEEE Working Conference on Software Visualization (VISSOFT)* (2020). 3, 6
- [WPJR11] WESSELS A., PURVIS M., JACKSON J., RAHMAN S. S.: Remote Data Visualization through WebSockets. In *Proc. of the 8th Int'l. Conf. on Information Technology: New Generations (ITNG)* (2011), pp. 1050–1051. URL: <https://doi.org/10.1109/ITNG.2011.182>, doi:10.1109/ITNG.2011.182. 7
- [WTM06] WANG Y., TEOH S. T., MA K.: Evaluating the Effectiveness of Tree Visualization Systems for Knowledge Discovery. In *Proc. of the Joint Eurographics - IEEE VGTC Symposium on Visualization (EuroVis)* (2006), pp. 67–74. URL: <https://doi.org/10.2312/VisSym/EuroVis06/067-074>, doi:10.2312/VisSym/EuroVis06/067-074. 10
- [WWF*13] WALLER J., WULF C., FITTKAU F., DOHRING P., HASSELBRING W.: Synchrovis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *Proc. of the 1st IEEE Working Conf. on Software Visualization (VISSOFT)* (2013), pp. 1–4. URL: <https://doi.org/10.1109/VISSOFT.2013.6650520>, doi:10.1109/VISSOFT.2013.6650520. 3
- [ZL15] ZHAO H., LU L.: Variational Circular Treemaps for Interactive Visualization of Hierarchical Data. In *Proc. of the IEEE Pacific Visualization Symposium (PacificVis)* (2015), pp. 81–85. URL: <https://doi.org/10.1109/PACIFICVIS.2015.7156360>, doi:10.1109/PACIFICVIS.2015.7156360. 3
- [ZZ01] ZIMMERMANN T., ZELLER A.: Visualizing memory graphs. In *Software Visualization* (2001), pp. 191–204. URL: https://doi.org/10.1007/3-540-45875-1_15, doi:10.1007/3-540-45875-1_15. 9

Chapter 6

User Guidance and User Behavior

6.1 Automatic Detection of Suspicious Time Windows

This section includes the paper [319] on how to detect suspicious time windows that hint at possible memory leaks or high memory churn.

Paper:

Markus Weninger, Elias Gander, Hanspeter Mössenböck:
Detection of Suspicious Time Windows in Memory Monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2019, Athens, Greece, October 21-22, 2019.

Detection of Suspicious Time Windows in Memory Monitoring

Markus Weninger
Institute for System Software
CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria
markus.weninger@jku.at

Elias Gander
CD Laboratory MEVSS
Johannes Kepler University
Linz, Austria
elias.gander@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University
Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

Modern memory monitoring tools do not only offer analyses at a single point in time, but also offer features to analyze the memory evolution over time. These features provide more detailed insights into an application's behavior, yet they also make the tools more complex and harder to use.

Analyses over time are typically performed on certain time windows within which the application behaves abnormally. Such suspicious time windows first have to be detected by the users, which is a non-trivial task, especially for novice users that have no experience in memory monitoring.

In this paper, we present algorithms to automatically detect suspicious time windows that exhibit (1) continuous memory growth, (2) high GC utilization, or (3) high memory churn. For each of these problems we also discuss its root causes and implications.

To show the feasibility of our detection techniques, we integrated them into AntTracks, a memory monitoring tool developed by us. Throughout the paper, we present their usage on various problems and real-world applications.

CCS Concepts • Software and its engineering → Software defect analysis; Software performance; • Mathematics of computing → Time series analysis.

Keywords Memory Monitoring, Automatic Time Window Detection, Memory Leak Analysis, Memory Churn Analysis

ACM Reference Format:

Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of Suspicious Time Windows in Memory Monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19)*, October

MPLR '19, October 21–22, 2019, Athens, Greece

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19)*, October 21–22, 2019, Athens, Greece, <https://doi.org/10.1145/3357390.3361025>.

21–22, 2019, Athens, Greece. ACM, New York, NY, USA, 10 pages.
<https://doi.org/10.1145/3357390.3361025>

1 Introduction

Modern programming languages such as Java use automatic garbage collection. Heap objects that are no longer reachable from so-called *GC roots* (e.g., from static fields or thread-local variables) are automatically reclaimed by a garbage collector (GC). Nevertheless, memory problems can still occur even in garbage-collected languages.

A *memory leak* [13] occurs if objects that are no longer needed remain reachable from GC roots due to programming errors. This leads to a *continuously growing memory consumption* which can cause the application to run out of memory, crashing it in the worst case [19].

Even though modern garbage collectors execute certain garbage-collection-related operations concurrently to the application [8, 12, 18], many garbage collection algorithms require *stop-the-world* pauses, i.e., the application is halted while the GC is running. Such GC phases can make up a significant portion of the application's run time.

A *high memory churn rate* stems from frequent unnecessary creation and collection of objects, also known as excessive dynamic allocations [40–42]. This leads to increased work for allocating these objects on the heap and an increased number of garbage collections, which can have a negative impact on an application's performance.

Such memory anomalies manifest themselves in various ways. They lead to different patterns in metrics such as memory consumption, GC frequency, or GC time. Inspecting and interpreting visualizations of these metrics, either in a tabular form or as time-series charts, can be hard for users, especially if they do not have a background in memory analysis.

The aim of this work is to ease the use of memory monitoring tools for novice users. To do so, we free users from the task of searching for different types of suspicious time windows by providing algorithms that automatically detect them. Thus, our contributions are:

1. different algorithms and heuristics to automatically detect suspicious time windows with
 - a. continuous memory growth (see Section 3.2).
 - b. high GC utilization (see Section 3.3).

- c. high memory churn (see Section 3.4).
including discussions on the root causes and the implications of the different types of memory anomalies.
- 2. a working implementation of our approach in the offline memory monitoring tool AntTracks Analyzer.

2 Background

AntTracks consists of two parts: The AntTracks VM [23–25], a virtual machine based on the Java Hotspot VM [47], and the AntTracks Analyzer [2, 51–56], a trace-based memory analysis tool. Since the techniques presented in this paper have been integrated into AntTracks, this section discusses memory traces and how AntTracks uses them.

2.1 Memory Snapshots versus Memory Traces

Many state-of-the-art tools use memory snapshots, i.e., heap dumps, for memory analysis, whereas AntTracks uses memory traces. While heap dumps may be sufficient for heap state analysis at a single point in time, it has been shown that they are not well suited for memory analysis over time [51]. This has various reasons. One of them is that heap dumps do not preserve object identities, i.e., one cannot distinguish whether two objects in two different heap dumps are the same or not.

Trace-based approaches try to circumvent the shortcomings of snapshots by continuously recording information while an application runs. Beside typical *memory traces* [5, 15, 16, 34, 35, 58, 61] that encode memory- and GC-related events such object allocations, object deaths, or object field accesses, there also other trace types such as *execution traces* that rather focus on call hierarchy information [6, 17, 46].

2.2 Trace Recording by the AntTracks VM

The AntTracks VM records memory events such as object allocations and object movements during GC by writing them into trace files [24]. Trace recording introduces a low run-time overhead of about 5%. Information about GC roots and the references between objects can also be added to the trace [23, 53]. To reduce the trace size, the AntTracks VM does not record any redundant data and applies compression [25].

2.3 AntTracks Analyzer

2.3.1 Reconstruction

The AntTracks Analyzer is able to parse a trace file by incrementally processing its events, which enables it to reconstruct the heap state for every garbage collection point [2]. A heap state is the set of heap objects that were live in the monitored application at a certain point in time. For every heap object, a number of properties can be reconstructed, including its address, its type, its allocation site, the heap objects it references, and the heap objects it is referenced by.

2.3.2 Analysis

The AntTracks Analyzer's core mechanism is object classification and multi-level grouping [54, 56] in which heap objects can be grouped according to certain criteria such as type, allocation site, allocating thread, and so on.

Various techniques to analyze the memory evolution over time have been presented in the past. For example, the approach described in [51, 52] detects the common problem of data structure growth. Handled incorrectly, data structures are often the root cause of memory leaks. In [55], we present an analysis technique that visualizes how the heap composition (i.e., the heap classified by a given object classifier combination) develops over time. All these approaches rely on a previously selected time window. By detecting suitable time windows automatically, such analysis techniques may be easier to apply for novice users.

3 Approach

This section explains how we support users to detect memory anomalies in an application's memory behavior. We present three different time window types, discuss their root causes and implications, and show heuristics and algorithms to automatically detect them. For each time window type, we also show an example on how AntTracks detects a suspicious time window in a real-world application and discuss how this window covers the problem's root cause.

3.1 Desired Window Characteristics

An ideal time window would outline just that portion of the program that should be investigated to find and remove the root cause of the underlying problem. Thus, we define characteristics that a detected time windows should exhibit.

Size Constraints First, detected time windows are desired to be *short* since one or more analysis techniques will be applied on the selected time window. The run time of most of these techniques depends on the number of garbage collections covered by the window. Despite this, windows should also cover a minimum number of garbage collections to prevent them from being only short, less important outliers.

Relevance The detected time windows should cover allocations and objects related to the underlying problem and as little noise, i.e., allocation and objects not related to the problem, as possible. If a window contained noise, e.g., allocations that are not relevant to a memory leak, the noise will also distort the results of analyses applied on the window. This makes it harder to reveal the root cause of the problem.

For example, a memory leak might manifest itself only after a certain point in time. Before that point, fluctuations in the memory can happen due to various reasons such as initialization procedures. These fluctuations are irrelevant for memory leak analysis, i.e., noise, and should be excluded from the detected time window.

Maximum Intensity Problems such as high garbage collection overhead generally do not persist throughout the whole application, but rather occur as *hotspots*. A detection algorithm should find the window that covers the most intense hotspot, e.g., the window with the highest overall garbage collection overhead.

Severity Since the aim of this work is to support novice users by automatically detecting *suspicious* time windows, every detection algorithm has to define thresholds to decide whether a detected time window is indeed suspicious. Windows that are not considered to be suspicious should not be presented to the users. For example, a detected garbage collection overhead hotspot may only be considered suspicious if the garbage collection overhead over the window exceeds a certain threshold, e.g., 10%.

3.2 Memory Leak Window

If a Java application contains a memory leak, certain objects are unintentionally kept alive, causing them to accumulate over time even though they are no longer needed. Consequently, the occupied heap space grows until the application runs out of memory, causing it to crash.

Unfortunately, such a growth trend may be difficult to recognize in a long running application's memory evolution, especially for novice users. Section 3.2.1 discusses the reasons for this in more detail. Thus, we present two algorithms to automatically detect time windows with suspicious growth trends, freeing the user from this task.

3.2.1 Trace Preprocessing

Detecting a memory-leak-induced growth trend based on the occupied heap memory can be difficult. First, the growth might be slow and only significant after the application has run for a long time. Additionally, the occupied memory fluctuates due to garbage collections, which makes it harder to see a clear trend. Finally, growth trends can be masked by *floating garbage*, that is, objects that are no longer reachable but have not been garbage collected yet. Thus, our approach detects growth trends based on the *reachable memory*, that is, the part of the heap memory that is reachable from *GC roots*. The reachable memory is unaffected by garbage collections and free from floating garbage which makes it the ideal basis to detect memory-leak-induced growth trends.

To calculate the reachable memory of a certain heap state, we start at the GC roots. By following all references recursively, we find all live objects on the heap. Summing up their sizes results in the amount of reachable memory, i.e., the memory in the heap that is alive.

This reachable memory calculation happens during trace file parsing, i.e., when the trace file is read for the first time. Calculating the reachable memory for every reconstructed heap state can slow down this parsing process. If performance is of concern to the user, AntTracks allows them to

enable sampled reachable memory calculation, i.e., to calculate the reachable memory only for certain heap states. When sampling is enabled, by default the reachable memory is calculated only for every second reconstructed heap state, i.e., at every other garbage collection, roughly cutting the time spent on reachable memory calculation in half. In our experience, this sampling frequency works well with the algorithms presented in Section 3.2.2. Nevertheless, users can adjust the sampling frequency to either reduce the parsing time or to increase the precision of the resulting reachable memory trend.

3.2.2 Automatic Time Window Detection

In the following, we present two algorithms to detect a time window with a growth trend in reachable memory. Both algorithms operate on a time series of reachable memory that was collected according to the preprocessing steps defined in Section 3.2.1.

Linear-regression-based Algorithm

This algorithm starts with an initial window that (1) includes the end of the application and (2) covers the last 10% of all garbage collection points. It performs a linear regression [30] over all reachable memory data points covered by the window and stores the slope of the linear regression line together with the current time window.

Next, the window is expanded to cover one more data point. Again, the algorithm performs a linear regression and stores the slope together with the time window. It continues in this way until the last time window, ranging from the application's start to the application's end, has been handled. Among all the stored windows, the one with the greatest regression line slope is chosen as the resulting time window. Finally, we also require that its regression slope is positive.

Figure 1 illustrates this algorithm. The plot shows 5 of the regression lines that would be calculated in the course of the algorithm. The regression line *E* has the greatest slope. Thus, the algorithm would return the window ranging from 1200 ms to 1500 ms. Note that due to limited space, we did not plot all regression lines.

Heuristic-based Algorithm

While the linear-regression-based algorithm is straightforward and easy to implement, the windows it detects do not always fulfill the criteria we defined in Section 3.1. This problem will be discussed in more detail in Section 3.2.3.

Considering the shortcomings of the linear-regression-based algorithm, we introduced a second time window detection algorithm. This algorithm mimics the way a human would search for a continuous memory growth: Find the longest time window over which the memory grows more-or-less continuously, allowing minor drops.

From the time series of reachable memory data points, the algorithm extracts the longest time window in which (1) the end of the application is included and (2) every data

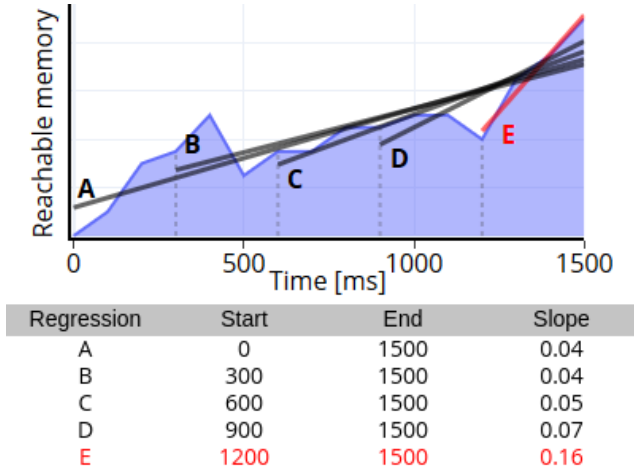


Figure 1. The linear-regression-based algorithm chooses the time windows with the steepest regression slope (E).

point within the window fulfills the growth condition. A data point fulfills the growth condition if its reachable memory is greater than that of the previous one. If this is not the case, a data point may still fulfill the growth condition if its reachable memory (1) is greater than the reachable memory of the window's first data point and (2) is at least 75% of the maximum reachable memory of all previous data points in the window. This heuristic requires a detected window to have an overall positive growth in reachable memory, but tolerates smaller drops.

The detection algorithm consists of the following steps. It starts with a window spanning only the application's first reachable memory data point. Next, it takes the second data point and tests whether it fulfills the growth condition with respect to the first point. If this is the case, the window is expanded up to this point. Otherwise, the current window is discarded and the algorithm starts again with a window spanning only the second data point. The algorithm continues in this way until the application's last data point has been handled. Finally, to make sure that the detected time window is long enough to be meaningful, the algorithm checks if the time window covers at least 10% of all garbage collections.

Figure 2 illustrates this algorithm. The initial time window starts at 0 ms and is expanded up to the first drop. Here the reachable memory drops by 50%. Thus, the current window is discarded and a new time window starts after the drop. From there on, the window can be expanded up to the end of the application because the second encountered drop is not strong enough.

Narrowing the Time Window In many applications, the reachable memory growth is not equally strong over the whole detected window. In such cases it is often possible to find a shorter subwindow with a higher growth rate. As stated in Section 3.1, a short time window is desired since subsequent analyses take less time to complete. Additionally,

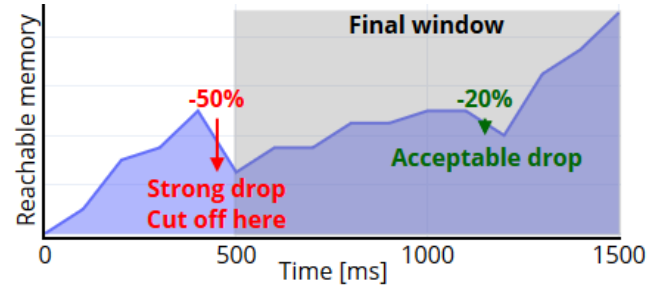


Figure 2. The heuristic-based algorithm detects the longest window without strong drops in the reachable memory.

if the growth over the shorter subwindow is caused by the memory leak, the problematic objects will stand out even more during the analysis. Consequently, the causes of the memory leak will be easier to recognize. Nevertheless, there is a chance that the strong growth covered by the shorter subwindow is actually unrelated to the memory leak, which instead manifests itself only in the slow and steady growth over the full time window. Thus, in AntTracks we decided to present both of these windows to the users. We leave it to their choice whether they want to perform a quick inspection of the strongest growth subwindow first.

When calculating a subwindow, we determine its minimum and maximum size. We define the minimum size as 10% of all reachable memory data points in the long window and the maximum size as 50% of all reachable memory data points in the long window. In any case, the minimum size must be at least two data points.

The algorithm takes the first data point in the long window as a starting point and builds all possible windows that (1) start at this data point, (2) end at another data point and (3) meet the size constraints. For all these windows, it then calculates the reachable memory growth per second and chooses the one with the quickest growth. This time window is remembered and the process is repeated with the next data point as starting point. This is repeated until all data points in the long window have been used as starting point. As a result, the algorithm remembered one window for each data point. Among all these windows, it again chooses the one with the quickest growth.

3.2.3 Examples

As already mentioned, the heuristic-based algorithm has been developed to overcome the flaws of the linear-regression-based algorithm which does not always fulfill the desired characteristics defined in Section 3.1. Figure 3 illustrates this problem. While both algorithms detect the same time window in the first two examples on the top half, in the third example the linear-regression-based algorithm includes a presumably irrelevant spike. The linear-regression-based algorithm performs even worse in the three examples on the bottom half of Figure 3. In these examples, it includes the

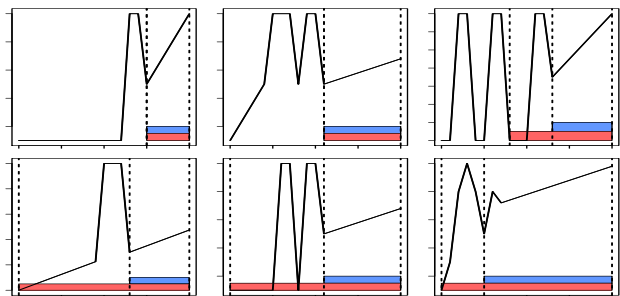


Figure 3. Six exemplary memory evolutions and the detected time windows (red = linear regression, blue = heuristic).

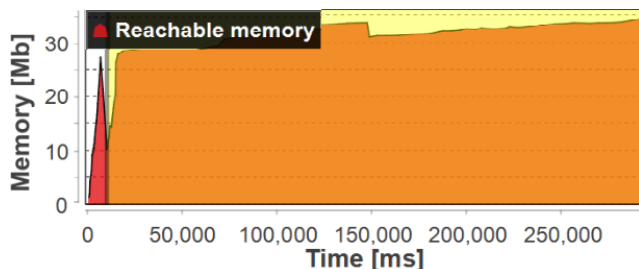


Figure 4. EasyTravel's memory evolution and the memory leak time window (yellow) detected by the heuristic-based algorithm.

whole application in the detected time window instead of just the final period of suspicious growth. Depending on the analysis technique that should be applied on the time window, the noise included in these time windows may make it difficult to recognize the root cause of the suspicious growth.

3.2.4 Case Study: Dynatrace EasyTravel

To show how our approach can be used in AntTracks, we apply it on the *Dynatrace easyTravel* application [11]. Dynatrace focuses on application performance monitoring (APM) and distributes *easyTravel* as their state-of-the-art demo application. It is a multi-tier application for a travel agency, using a Java backend. An automatic load generator can simulate accesses to the service. When *easyTravel* is started, different problem patterns can be enabled and disabled, one of which is a hidden memory leak somewhere in the backend.

Figure 4 shows the memory evolution of the application and the time window that has been detected automatically using the heuristic-based memory leak time window detection algorithm. After an initial peak (which is not included in the final time window), the memory mostly grows, except for a drop at around 150,000ms, which is small enough to be tolerated by the algorithm.

This time window can be inspected with different analysis techniques. For example, the time window could be checked for growing data structures, as done in [51]. The less noise the window contains, the easier it becomes for users to spot those data structures that are involved in the memory leak.

3.3 High GC Overhead Analysis

The *garbage collection overhead* is the ratio between the time spent on garbage collections and the application's overall run time. The duration of a garbage collection partly depends on the number of surviving objects. The more objects survive, the more moves have to be executed by the garbage collector. This leads to increased garbage collection times.

To reduce an application's garbage collection overhead, users should inspect the time window that exhibits the highest GC overhead. In this time window, analysis techniques to identify those objects that survive and thus slow down the collections could be applied. Yet, according to our experience, most novice users disregard problematic garbage collector behavior and only focus on the memory evolution when inspecting an application. Thus, we support them by detecting the time window with the highest GC overhead automatically.

3.3.1 Time Window Detection

On the other hand, windows that cover a very large number of garbage collections might take too long to analyze. They also do not provide more insight because a shorter window will still reveal the reasons for the high garbage collection overhead. Thus, algorithms should only look for windows that cover at least 5 garbage collections and do not cover more than 50 garbage collections. The numbers used as smallest and largest window size have proven to work well in most scenarios.

We assume that we know the start time and the end time of each garbage collection in the application. To find the window with the highest garbage collection overhead that meets the window size constraint, our algorithm performs the following steps: First, it selects a start timestamp for the time window. In the first iteration, the start timestamp is the trace's very first timestamp, i.e., the timestamp that marks the start of the application. It then builds all windows that (1) start at this timestamp, (2) meet the size constraint, and (3) end at the end of a garbage collection. Figure 5 demonstrates the first iteration of the algorithm where windows A to E are built from the initial timestamp. For each of these windows, the garbage collection overhead is calculated by dividing the time spent in garbage collections over the window by the duration of the window. Among all windows constructed this way, the algorithm remembers the one with the highest garbage collection overhead. In the example in Figure 5, this would be window C.

In the following iterations, the start timestamp moves forward such that every end of a garbage collection serves as start timestamp once. For every start timestamp then again all valid windows are built and the one with the highest garbage collection overhead is remembered. Figure 6 shows the remembered window for each start timestamp (windows ① to ⑤). Among these windows, the algorithm

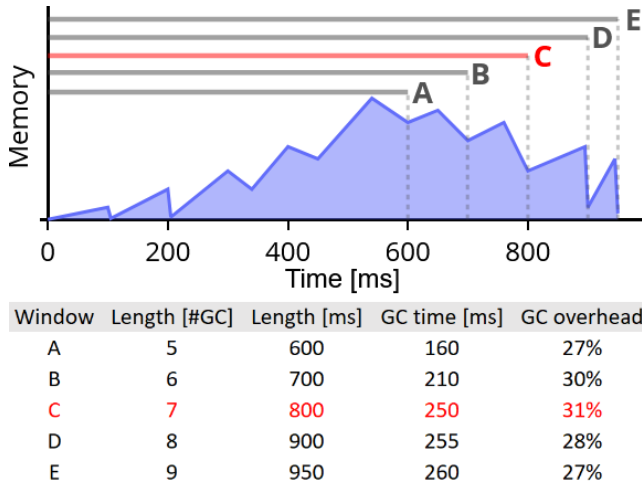


Figure 5. Windows A-E are all the valid windows that start at the first timestamp. Window C is the one with the highest garbage collection overhead.

finally chooses the one with the highest overhead. This final window has the highest overhead of all possible windows that meet the size constraint. In the example in Figure 6, this would be window ③.

The final window is only accepted if it has a overhead of at least 10%. This threshold prevents us from detecting a window with a generally low garbage collection overhead.

3.3.2 Case Study: AntTracks

We applied AntTrack’s garbage collection overhead window detection mechanism on AntTracks itself. As shown in Figure 7, it was able to detect the time window with the highest garbage collection overhead that meets the window size constraints. The time window covers the most intense part of a garbage collection overhead hotspot. By analyzing which objects had to be moved by the GC most often during this window, we were able to find and fix a bottleneck involving `long[]` objects that were created in AntTracks when pointer information was read from trace files.

3.4 High Memory Churn Analysis

As we have seen before, garbage collections are slower the more objects survive. Analogously, they are fast when many objects die. Yet, even these fast garbage collections have to pause the application. These *stop-the-world pauses* require all application threads to halt at so-called safepoints, i.e., at specific instructions that block the executing thread if necessary, before the GC can start to work.

Even though modern garbage collectors such as Shenandoah [12] or the Z GC [48, 49] perform certain operations concurrently to the running application, nearly all garbage collectors still have to use stop-the-world pauses at some points. When many garbage collections occur over a short time span, these pauses can lead to a significant overhead.

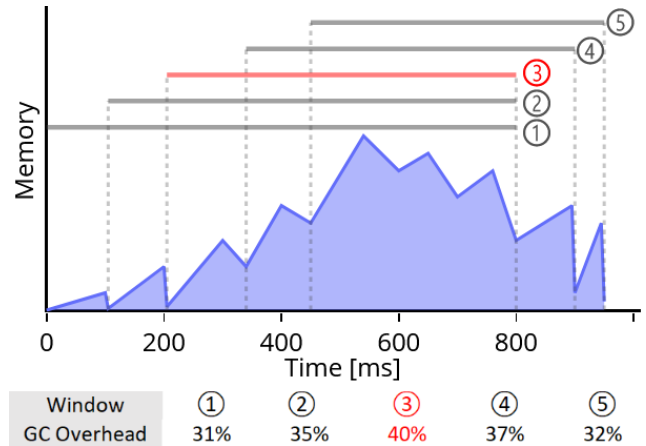


Figure 6. Windows ①-⑤ are the windows with the highest garbage collection overhead for each start timestamp. Window ③ is the one with the highest overhead overall.

A typical cause for frequent garbage collections are objects that are allocated in great numbers and turn into garbage shortly after their allocation.

In the next section, we present an algorithm that detects suspicious time windows based on an application’s *memory churn rate*. The memory churn rate is the frequency at which the application discards memory within a certain time window. This hints at a wasteful use of objects, i.e., an unnecessarily high amount of short-living object allocations. Often, algorithms can be adjusted to use fewer temporary objects, which leads to two improvements: (1) Less time is spent for the allocation of objects, and (2) the number of garbage collections is reduced. A common case for the excessive use of temporary objects in Java is the boxing of primitives.

3.4.1 Time Window Detection

To detect the time window with the highest memory churn rate we basically use the algorithm described in Section 3.3.1. The only difference is that this time the algorithm searches for the window with the highest memory churn rate instead

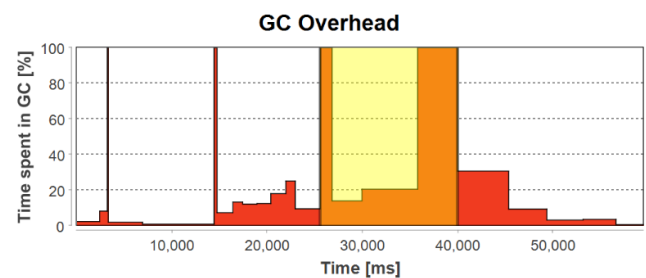


Figure 7. Automatically detected GC overhead window in AntTracks, highlighted in yellow from around 26,000ms to 40,000ms.

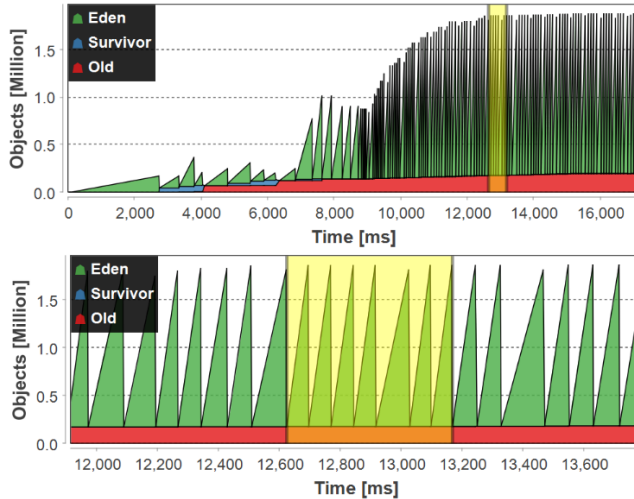


Figure 8. Automatically detected time window with high memory churn in the `finagle-http` benchmark (global view and zoomed-in view).

```

1 val response: Future[http.Response] = client(request)
2 for (i <- 0 until NUM_REQUESTS) {
3   Await.result(response.onSuccess { rep: http.Response =>
4     totalLength += rep.content.length })}

```

Listing 1. Problematic part of the method `FinagleHttp.runIteration`.

of the highest garbage collection overhead. To do so, it calculates the churn rate of a window by dividing the total number of bytes freed within the window by the window’s duration. To calculate the number of freed bytes for a given garbage collection, all we need to know is the size of the heap before the collection and after the collection.

3.4.2 Example: `Finagle-http`

Renaissance [33] is a benchmark suite composed of modern, real-world, concurrent, and object-oriented workloads that exercise various concurrency primitives of the JVM. Since this benchmark suite is rather new, it has not yet been the subject of a memory study [26]. Thus, it is perfectly suited to test whether AntTracks is able find memory problems in real-world applications unknown to the inspector.

First, we downloaded the benchmark suite¹ in version 0.9 and created a trace file of every benchmark. Then, we loaded these trace files into AntTracks and inspected the automatically detected time windows. One benchmark that attracted our attention was `finagle-http`. According to the benchmark’s documentation, it *sends many small Finagle HTTP requests to a Finagle HTTP server and awaits response*. This benchmark exhibits a high memory churn. Its memory evolution and the automatically detected memory churn window can be seen in Figure 8.

¹Renaissance benchmark suite: <https://renaissance.dev/>

```

1 val response: Future[http.Response] = client(request)
2 val h = { rep: http.Response => totalLength += rep.content.length }
3 for (i <- 0 until NUM_REQUESTS) Await.result(response.onSuccess(h))

```

Listing 2. Fixed version of the method `FinagleHttp.runIteration`.

Inspecting this memory churn using AntTracks’s short-lived objects analysis feature revealed that the type `FinagleHttp$$anonfun$runIteration$1$$...$` has a high churn rate. The naming pattern reveals that these are Scala objects, more specifically, anonymous functions, which are allocated in the method `runIteration` of the benchmark’s main class `FinagleHttp`. Since such a rapid allocation and collection of anonymous functions is unlikely to be intentional, we looked up the method’s source code. The problematic part can be seen in Listing 1. In the loop, a lot of anonymous function objects are created, waiting for an HTTP request to succeed to increment the counter `totalLength`. Listing 2 shows our fix for this problem. Only a single response handler is created which is reused for every HTTP request. This fix reduces the overall amount of allocated temporary objects by about 25%.

3.5 Window Detection Performance

The complexity of all algorithms is $O(n)$, where n is the number of garbage collections covered by the trace file. For example, applied on the trace of the Dynatrace EasyTravel application that has been shown in Section 3.2.4, which covers about 700 garbage collections, the different time window detection algorithms take between 5ms and 30ms on average. Thus, our algorithms can scale up to process data reconstructed from traces that cover thousands of garbage collections.

4 Related Work

Memory Leak Detection To support memory leak detection, various approaches and tools have been developed over the last years. Sor and Srirama [45] classify these approaches into the following groups:

1. *Online approaches* that actively monitor and interact with the running virtual machine, separated into approaches that
 - a. *measure staleness* [3, 14, 35, 36, 38, 59]. The longer an object is not used, the staler it becomes. Staleness analysis tries to reveal objects that do not get collected by the GC but become stale, since they are most likely to be leaking. The challenge these approaches face is that tracking object accesses is extremely expensive.
 - b. *detect growth* [4, 20, 21, 43, 44]. These approaches group the live heap objects (usually either by their types or allocation sites) and detect growth using various metrics.

2. *Offline approaches* that collect information about an application for later analysis, separated into approaches that
 - a. *analyze heap dumps and other kinds of captured state* [22, 27–29]. Compared to online approaches, offline approaches often perform more complicated analyses based on the object reference graph, involving graph reduction, graph mining and ownership analysis.
 - b. *use visualization* to aid leak detection [7, 31, 37].
 - c. *employ static source code analysis* [9, 60].
3. *Hybrid approaches* that combine online features as well as offline features [10, 39, 58].

In this taxonomy, AntTracks would be classified as a *hybrid approach*. It collects detailed memory traces *online* using its VM, while the processing of these traces happens *offline*. Its offline analysis tool mostly focuses on the visualization and automatic detection of heap growth.

Memory Churn Analysis For example, Peiris and Hill [32] presented EMAD, the *Excessive Memory Allocation Detector*. Compared to AntTracks, which detects memory churn offline using memory traces, EMAD uses dynamic binary instrumentation and exploratory data analysis to determine whether an application performs excessive dynamic memory allocations.

5 Limitations

One limitation of our work is its currently limited evaluation based on a small set of use cases. We plan to find more open-source projects that suffered from memory anomalies in the past which we can use to build a reference set of real-world applications. This collection could then be used to evaluate memory monitoring tools.

In addition to that, we are currently conducting a user study. One goal of this study is to see how well people with different backgrounds are able to detect suspicious time windows themselves. Preliminary results suggest that novice users are not always able to recognize suspicious memory growth in an application. Also, it seems that most users also underestimate the possible severity of high memory churn.

6 Future Work

User Study As stated in the previous section, we are currently conducting a user study with university students having various levels of expertise. The aim of the study is to gain insights on how well the study participants are able to analyze memory anomalies with AntTracks and which features they would expect from a memory monitoring tool in general. This could help the community to improve the quality of memory monitoring tools.

Visualization Many of AntTracks’s analysis features communicate their results in form of tables, line charts or stacked

area charts. We plan to evaluate alternative visualization approaches, for example, an application’s memory evolution could also be displayed as *small multiples* [50] or as a *software city* [57]. To investigate keep-alive relationships in a heap state, we plan to support users by displaying aggregated heap objects as graphs [1].

Guided Exploration The aim of this work is to automate the first step of memory evolution analysis over time, namely the selection of a suspicious time window. Nevertheless, once a time window is selected, an appropriate analysis approach has to be selected, and the users are left on their own during this analysis. Thus, we are currently integrating features into AntTracks that we call *guided exploration*. The goal of guided exploration is to lead users through AntTracks’s different analysis views. Within each view, those parts that contain the most information should automatically be detected, highlighted and explained to the user. This way, we want to achieve a learning-by-doing effect, with the goal that AntTracks should be usable by users without any prior memory monitoring experience.

7 Conclusion

In this paper, we presented an approach to automatically detect time windows that show typical behaviors of various memory anomalies. Freeing users from this non-trivial task enables them to focus more on finding the root cause of the problem. Especially inexperienced users that often struggle to recognize anomalies on their own profit from this feature.

The first type of memory anomaly that our approach is able to automatically detect is *continuous memory growth* caused by memory leaks.

The second type are windows that suffer from *high GC overhead*. High GC overhead can stem from two main root causes. Either the individual garbage collections within the time window took a long time, or a very high number of garbage collections had to be performed.

The third type of suspicious time windows we are able to detect are those that show *high memory churn*. High memory churn is caused by objects that are frequently allocated and freed shortly after their allocation. This also leads to a high number of garbage collections.

These algorithms to automatically select suspicious time windows have been integrated into AntTracks, a memory monitoring tool developed by us. Throughout the paper, their applicability was shown by applying them to different real-world applications.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

- [1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proc. of the 5th Int'l Symp. on Software Visualization (SOFTVIS '10)*.
- [2] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proc. of the Principles and Practices of Programming on The Java Platform (PPPJ '15)*.
- [3] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: Bit-encoding Online Memory Leak Detection. In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*.
- [4] K. Chen and J. Chen. 2007. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Proc. of the 31st Annual Int'l Computer Software and Applications Conf. (COMPSAC '07)*.
- [5] Trishul Chilimbi, Richard Jones, and Benjamin Zorn. 2000. Designing a Trace Format for Heap Allocation Events. In *Proc. of the 2nd Int'l Symposium on Memory Management (ISMM '00)*.
- [6] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. 2008. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software* 81, 12 (2008), 2252 – 2268.
- [7] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '99)*.
- [8] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proc. of the 4th Int'l Symposium on Memory Management (ISMM '04)*.
- [9] Dino Distefano and Ivana Filipović. 2010. Memory Leaks Detection in Java by Bi-abductive Inference. In *Proc. of the Int'l Conf. on Fundamental Approaches to Software Engineering (FASE 2010)*.
- [10] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. 2007. Blended Analysis for Performance Understanding of Framework-based Applications. In *Proc. of the 2007 Int'l Symposium on Software Testing and Analysis (ISSTA '07)*.
- [11] Dynatrace. 2019. Demo Applications: easyTravel. <https://community.dynatrace.com/community/display/DL/Demo+Applications+-+easyTravel>
- [12] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *Proc. of the 13th Int'l Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.
- [13] Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Proc. of the 40th Int'l Conf. on Software Engineering: Comp. Proc. (ICSE '18)*.
- [14] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proc. of the 11th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*.
- [15] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanović. 2002. Error-free Garbage Collection Traces: How to Cheat and Not Get Caught. In *Proc. of the 2002 ACM SIGMETRICS Int'l Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*.
- [16] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. 2006. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (May 2006), 476–516.
- [17] Swaminathan Jayaraman, Bharat Jayaraman, and Demian Lessa. 2017. Compact Visualization of Java Program Execution. *Software: Practice and Experience* 47, 2 (2017), 163–191.
- [18] Richard Jones, Antony Hosking, and Eliot Moss. 2016. *The garbage collection handbook: the art of automatic memory management*. Chapman and Hall/CRC.
- [19] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '07)*.
- [20] Maria Jump and Kathryn S. McKinley. 2007. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proc. of the 34th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL '07)*.
- [21] Maria Jump and Kathryn S. McKinley. 2009. Detecting Memory Leaks in Managed Languages with Cork. *Software: Practice and Experience* 40, 1 (2009).
- [22] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In *Proc. of the ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD '10)*.
- [23] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proc. of the 13th Int'l. Conf. on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '16)*.
- [24] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proc. of the 6th ACM/SPEC Int'l. Conf. on Performance Engineering (ICPE '15)*.
- [25] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proc. of the 7th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '16)*.
- [26] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.
- [27] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *Proc. of the 20th European Conf. on Object-Oriented Programming (ECOOP '06)*.
- [28] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP '03)*.
- [29] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. In *Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications (OOPSLA '07)*.
- [30] Raymond H Myers and Raymond H Myers. 1990. *Classical and modern regression with applications*. Vol. 2. Duxbury press Belmont, CA.
- [31] Wim De Pauw and Gary Sevitsky. 2000. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency: Practice and Experience* 12, 14 (2000).
- [32] Manjula Peiris and James H. Hill. 2016. Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern. In *Proc. of the 7th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '16)*.
- [33] Aleksandar Prokopec, Andrea Rosà, David Leopoldseger, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2019)*.
- [34] J. Qian and X. Zhou. 2012. Inferring weak references for fixing Java memory leaks. In *Proc. of the 2012 28th IEEE Int'l Conf. on Software Maintenance (ICSM '12)*.
- [35] Derek Rayside and Lucy Mendel. 2007. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proc. of the 22nd IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '07)*.

- [36] Derek Rayside, Lucy Mendel, and Daniel Jackson. 2006. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proc. of the Int'l Workshop on Dynamic Systems Analysis (WODA '06)*.
- [37] S. P. Reiss. 2009. Visualizing The Java Heap to Detect Memory Problems. In *Proc. of the 5th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '09)*.
- [38] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. 2001. Heap Profiling for Space-efficient Java. In *Proc. of the ACM SIGPLAN 2001 Conf. on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 104–113. <https://doi.org/10.1145/378795.378820>
- [39] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. 2000. Automatic Removal of Array Memory Leaks in Java. In *Proc. of the 9th Int'l Conference on Compiler Construction (CC '00)*.
- [40] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proc. of the 2nd Int'l Workshop on Software and Performance (WOSP '00)*.
- [41] Connie U. Smith and Lloyd G. Williams. 2002. New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*
- [42] Connie U. Smith and Lloyd G. Williams. 2003. More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Intl. CMG Conf.*
- [43] V. Sor, P. Oü, T. Treier, and S. N. Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *Proc. of the 2013 IEEE Int'l Conf. on Software Maintenance (ICSM '13)*.
- [44] Vladimir Šor, Nikita Salnikov-Tarnovski, and Satish Narayana Srirama. 2011. Automated Statistical Approach for Memory Leak Detection: Case Studies. In *On the Move to Meaningful Internet Systems (OTM 2011)*.
- [45] Vladimir Šor and Satish Narayana Srirama. 2014. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software* 96 (2014).
- [46] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. 2005. Extracting sequence diagram from execution trace of Java program. In *Proc. of the 8th Int'l Workshop on Principles of Software Evolution (IWPESE '05)*.
- [47] Oracle. 2019. The HotSpot Group. <http://openjdk.java.net/groups/hotspot/>
- [48] Oracle. 2019. ZGC - The Z Garbage Collector. <http://openjdk.java.net/projects/zgc/>
- [49] Per Lidén & Stefan Karlsson. 2018. The Z Garbage Collector - An Introduction, FOSDEM 2018. <http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf>
- [50] Stef van den Elzen and Jarke J. van Wijk. 2013. Small Multiples, Large Singles: A New Approach for Visual Data Exploration. *Comput. Graph. Forum* 32 (2013).
- [51] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proc. of the 2019 ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '19)*.
- [52] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.
- [53] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proc. of the 15th Int'l Conf. on Managed Languages & Runtimes (ManLang '18)*.
- [54] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proc. of the 8th ACM/SPEC on Int'l Conf. on Performance Engineering (ICPE '17)*.
- [55] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Companion of the 2019 ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '19)*.
- [56] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proc. of the 9th ACM/SPEC Int'l Conf. on Performance Engineering (ICPE '18)*.
- [57] Richard Wettel and Michele Lanza. 2007. Visualizing Software Systems as Cities. In *Proc. of the 4th IEEE Int'l Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '07)*.
- [58] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI '11)*.
- [59] Guoqing Xu and Atanas Rountev. 2008. Precise Memory Leak Detection for Java Software Using Container Profiling. In *Proc. of the 30th Int'l Conf. on Software Engineering (ICSE '08)*.
- [60] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *Proc. of the Annual IEEE/ACM Int'l Symposium on Code Generation and Optimization (CGO '14)*.
- [61] H. Yu, X. Shi, and W. Feng. [n.d.]. LeakTracer: Tracing leaks along the way. In *Proc. of the 2015 IEEE 15th Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '15)*.

6.2 Cognitive Walkthrough and User Study

This section includes the paper [321] on our usability assessment of AntTracks as well as our user study with novices in memory analysis.

Paper:

Markus Weninger, Paul Grünbacher, Elias Gander, Andreas Schörgenhumer: *Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study*. In *Proceedings of the ACM on Human Computer Interaction*, Vol. 4 (EICS), June 2020.

Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study

MARKUS WENINGER, Institute for System Software, Johannes Kepler University Linz, Austria

PAUL GRÜNBACHER, Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria

ELIAS GANDER, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

ANDREAS SCHÖRGENHUMER, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

Memory analysis tools are essential for finding and fixing anomalies in the memory usage of software systems (e.g., memory leaks). Although numerous tools are available, hardly any empirical studies exist on their usefulness for developers in typical usage scenarios. Instead, most evaluations are limited to reporting performance metrics. We thus conducted a study to empirically assess the usefulness of the interactive memory analysis tool AntTracks Analyzer. Specifically, we first report findings from assessing the tool using a cognitive walkthrough, guided by the Cognitive Dimensions of Notations Framework. We then present the results of a qualitative user study involving 14 subjects who used AntTracks to detect and resolve memory anomalies. We report lessons learned from the study and implications for developers of interactive memory analysis tools. We hope that our results will help researchers and developers of memory analysis tools in defining, selecting, and improving tool capabilities.

CCS Concepts: • **General and reference** → **Evaluation**; Metrics; Performance; • **Human-centered computing** → **User studies**; **Usability testing**; **Walkthrough evaluations**; **Empirical studies in HCI**; *Graphical user interfaces*; *User centered design*; • **Information systems** → Users and interactive retrieval; • **Software and its engineering** → Software performance.

Additional Key Words and Phrases: Interactive Memory Analysis Tools; Cognitive Walkthrough; Cognitive Dimensions; User Study; Usefulness; Usability; Utility; Assessment

ACM Reference Format:

Markus Weninger, Paul Grünbacher, Elias Gander, and Andreas Schörghener. 2020. Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 75 (June 2020), 37 pages. <https://doi.org/10.1145/3394977>

1 INTRODUCTION

Interactive memory analysis tools collect, process, transform, and visualize information about the memory footprint of software systems. Snapshot-based tools analyze a single point in time while trace-based tools allow users to explore a period of time [109]. For example, many tools present the heap state of an application as a type histogram displaying the number of objects and bytes allocated for each type. Analyzing such information allows users to detect potential memory anomalies and to reveal their root cause.

Authors' addresses: Markus Weninger, markus.weninger@jku.at, Institute for System Software, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Paul Grünbacher, paul.gruenbacher@jku.at, Institute for Software Systems Engineering, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Elias Gander, elias.gander@jku.at, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Andreas Schörghener, andreas.schoergener@jku.at, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Human-Computer Interaction*, <https://doi.org/10.1145/3394977>.

Existing interactive memory analysis tools provide a variety of capabilities to analyze different aspects of memory usage. For example, the Eclipse Memory Analyzer (MAT) [29] and VisualVM [91] are the most commonly used open source memory analysis tools for Java. While MAT purely focuses on memory analysis, VisualVM is a more general performance analysis tool including advanced memory analysis features. Kieker [58, 112, 113] is a well-known general performance framework for monitoring and analyzing the run-time behavior (including memory) of concurrent or distributed software systems. Well-known examples of commercial tools providing memory analysis features are the Dynatrace application performance monitoring (APM) platform [28] and the JProfiler [30], which offers memory profiling and a heap walker for Java applications.

So far, most memory analysis tools have been evaluated with a focus on their performance overhead and feasibility while only little empirical research exists on their *usefulness* in practical environments and for realistic usage scenarios. The term usefulness captures a tool's utility, i.e., to what degree it allows users to achieve their goals, and its usability, i.e., how well users can make use of the offered features. The study by Zaman et al. [130] is an exception in the field of performance engineering, as the authors show for two enterprise systems that test-based performance analyses need to be complemented with user-centric assessments to better understand user-perceived quality. The authors strongly argue that performance engineering should use the knowledge on how to conduct user-centric analysis from other fields.

This paper thus reports findings of a cognitive walkthrough to assess the usability of an interactive memory analysis tool. We also conducted a qualitative study to analyze the behavior of users analyzing memory anomalies in a realistic context. We performed our research using *AntTracks*, a memory monitoring system which comprises the AntTracks VM [65–67], a custom virtual machine based on the Java Hotspot VM [87], and the AntTracks Analyzer [7, 116–119, 121, 122, 124], a trace-based memory analysis tool. The AntTracks VM records memory events such as object allocations and object movements during garbage collection (GC) by writing them into trace files [65–67]. The AntTracks Analyzer then parses trace files by incrementally processing these events, thereby allowing to reconstruct the heap state for every GC point [7]. Various memory analyses can be performed with AntTracks, including heap state analysis [116, 121, 124], data structure growth analysis [117, 119], and heap evolution visualization [122].

Specifically, the contributions of our work encompass (1) a discussion of common memory analysis activities and tool capabilities based on existing research and tools (Section 3), (2) a realization of these capabilities in the AntTracks Analyzer memory analysis tool (Section 4), including an assessment based on a cognitive walkthrough following the Cognitive Dimensions (CD) of Notations Framework (Section 5), (3) the design (Section 6) and results (Section 7) of a usefulness study involving 14 participants who used AntTracks in two realistic analysis scenarios, and (4) general recommendations for researchers and developers of interactive memory analysis tools we derived from lessons learned during the study (Section 8) as well as a discussion on how we used these recommendations to further improve AntTracks (Section 9). Section 10 discusses threats to validity and Section 11 concludes the paper.

2 RESEARCH METHOD

The field of human-computer interaction (HCI) distinguishes inspection-based and test-based approaches [47] to evaluate the usability of software systems. Inspection-based techniques aim at assessing and improving interactive systems by checking them against some standard, such as Nielsen's usability attributes [82] or the Cognitive Dimensions (CD) of Notations Framework [8, 9, 11, 39–41]. Test-based techniques, on the other hand, involve end users in the evaluation.

Memory analysis is a highly complex and interactive process. Thus, our research method relies on both inspections and testing. Specifically, we investigated two research questions on the usefulness of interactive memory analysis tools using the example of AntTracks: (RQ1) Regarding *usability* we assessed AntTracks' memory analysis capabilities from the perspective of software engineers, guided by the CD framework and Nielsen's usability attributes. (RQ2) Regarding *utility* we conducted a user study analyzing a real-world Java web application with seeded memory defects. Based on the results and lessons learned, we synthesized recommendations intended to support developers of interactive memory analysis tools. We conducted our research in four steps:

Identification of Memory Analysis Activities. We studied related research and features of state-of-the-art memory analysis tools to identify important memory analysis activities benefiting from tool support. In addition to that, we present how these memory analysis activities manifest themselves in the memory monitoring tool AntTracks, the main subject of this study.

Cognitive Walkthrough and Tool Improvement. To assess AntTracks' usability, we first performed a cognitive walkthrough of the identified activities using the CD framework, which offers a vocabulary for discussing usability issues and their trade-offs. The CD framework has been used successfully to assess software tools [5, 61, 72, 94, 95], visual diagrams [10], temporal specification notations [63, 64], or visual modeling languages [22, 125]. Table 1 shows a summary of these dimensions. A detailed description of the framework and the cognitive dimensions can be found online [40]. The primary aim of this cognitive walkthrough was to reveal and fix possible usability flaws before conducting the user study and to define the scope for the user study.

User Study. We designed our study based on the findings from the CD assessment and the guidelines for conducting empirical studies by Runeson and Höst [101]. Software engineering students from our university used AntTracks to investigate the memory evolution of an application to detect anomalies such as memory leaks or high memory churn (cf. Section 6). For each anomaly, the participants aimed at revealing its root cause using the memory analysis. During this process, we asked each study participant to 'think aloud' [47], i.e., to describe what they were doing and to comment on any concerns. The participants were interviewed on the utility of the tool [23] and also completed a usability questionnaire [82].

Table 1. Cognitive dimensions used for the walkthrough (taken from [40]).

Dimension	Description
Abstraction	types and availability of abstraction mechanisms
Closeness of Mapping	closeness of representation to domain
Consistency	similar semantics are expressed in similar syntactic forms
Diffuseness	verbosity of language
Error-proneness	notation invites mistakes
Hard Mental Operations	high demand on cognitive resources
Hidden Dependencies	important links between entities are not visible
Premature Commitment	constraints on the order of doing things
Progressive Evaluation	work-to-date can be checked at any time
Provisionality	degree of commitment to actions or marks
Role-expressiveness	the purpose of a component is readily inferred
Secondary Notation	extra information in means other than formal syntax
Viscosity	resistance to change
Visibility	ability to view components easily

Derivation of Implications. Finally, we synthesized recommendations and lessons learned based on the detailed results and feedback obtained from the study. In addition, we discuss how these recommendations were used to further improve AntTracks.

3 MEMORY ANALYSIS ACTIVITIES

We present key activities supported by interactive memory analysis tools based on our experiences and related work. We focus on memory analysis for managed languages such as Java or C#. We will show that the tools vary regarding their support, e.g., some tools only visualize raw data and leave the analysis to the users, while other tools automate certain analyses activities.

3.1 Collecting Memory Data

Basic tools for snapshot-based inspections of memory usage mostly rely on heap dumps, which can be created by tools such as HPROF [88, 92] or jmap [86]. The following techniques are used for analyzing more specific details of snapshots or memory usage over time [6]: (1) A *modified execution environment* such as a custom Java VM that can access internal information; (2) a *sampling-based approach*, e.g., an agent using the Java VM Tool Interface [90] to receive periodical callbacks about memory-relevant events in the application; or (3) an *instrumentation-based approach* that relies on adding code to an existing application, either before compilation (e.g., AspectJ [57]) or at run time (e.g., ASM [14, 15, 62] or Javassist [17, 18]).

3.2 Detection of Memory Anomalies

Before inspecting an application in detail, memory analysis tools support users in detecting memory anomalies such as memory leaks, high memory churn, memory bloat, or unusual GC behavior.

3.2.1 Memory Leaks.

Memory leaks [35] in managed languages occur if objects no longer needed remain reachable from garbage collection roots (e.g., static fields or local variables) due to programming errors. For example, objects may accumulate over time when a developer forgets to remove them from long-living data structures [117]. Such leaks lead to a growing memory footprint, which at some point will cause an application to crash. There are two main approaches to detect memory leaks: (1) Techniques detecting *staleness* [12, 44, 96, 128] assume that objects not used for a long time are likely involved in a memory leak. However, the proposed techniques are hardly used outside academia due to their high costs of tracking objects. (2) Techniques detecting *growth* [16, 54, 78, 108] are thus still the de-facto standard in state-of-the-art memory analysis tools and mostly rely on users interpreting visualizations. For example, VisualVM [91] periodically plots the memory footprint in a time-series chart. The user then has to check for suspicious sections of continuous growth that might hint at a memory leak. Similarly, JConsole [85] can read a running application's Java Management Beans to plot the currently occupied heap memory separated by eden space, survivor space, and old space.

3.2.2 Memory Churn.

Memory churn occurs when large numbers of short-living objects are created by an application, thereby causing many garbage collections. Such excessive dynamic allocation behavior [106] typically has a negative impact on performance. However, obtaining the information on how long objects survive before dying is expensive [45, 99, 100]. Most tools are thus limited to analyzing the number of allocations, but not the exact lifetime of objects. Objects frequently allocated in bursts typically do not survive for a long time and thus the high allocation rate already indicates memory churn. Memory churns can be detected either by visually spotting spike patterns in memory charts (i.e., high consumption of memory followed by many object deaths) or by plotting the number of

allocations over time (i.e., detecting allocation-intensive time windows), as for example done in Dynatrace [28] or Kieker [58, 112, 113].

3.2.3 *Memory Bloat.*

Memory bloat [52, 77, 127] describes the inefficient use of memory for achieving seemingly simple tasks. It is often caused by heavily using (object-oriented) abstractions such as over-generalized data structures. Most techniques for detecting memory bloat thus focus on analyzing data structures requiring many auxiliary objects [79] or the inefficient usage of data structures operations for adding, getting, or removing elements [126, 129].

3.2.4 *Unusual GC Behavior.*

The behavior of the GC can also indicate memory problems. Instead of looking at the memory behavior of an application, this anomaly is detected by inspecting the garbage collector, e.g., by measuring GC overhead via the garbage collection count and the garbage collection duration [68].

3.3 Inspection of Memory Anomalies

Once a suspicious memory behavior is detected, the user can inspect a single point in time or a time interval to reveal the root cause of the problem.

3.3.1 *Single Point in Time.*

The most common technique is a *heap state analysis*, which relies on reconstructing the objects that were alive at a certain point in time. For every object on the heap, a number of properties can be reconstructed depending on the tool: these may include the object's address, its type, its allocation site, the heap objects it references, the heap objects it is referenced by, the thread allocating the object, and a list of root pointers referencing it. Users can then examine (groups of) objects on the heap or study metrics about the heap state. *Object-based techniques* allow to inspect heap objects in a bottom-up or top-down fashion [116]. In the bottom-up approach the user searches for big object groups (e.g., objects of the same type) and then tries to free them. The most common visualization to find these object groups is a type histogram grouping all heap objects by their types, and also showing the memory occupied by each type. The object type(s) consuming most memory can then be inspected in detail. Some tools support users by displaying the path to the GC roots, while other approaches assist users by displaying the code that has allocated the objects. Visualization approaches [2, 46, 73, 76, 97, 102, 131] aggregating the object graph (e.g., based on its dominator tree [69, 75, 116]) are useful to analyze the heap's composition. A user following the top-down approach first selects a GC root or a heap object that keeps alive many other objects. The user then inspects the objects reachable from this root or object and searches for possible cut points in the path [116, 117]. *Metric-based techniques* derive metrics from the heap state that allow to analyze the heap state by revealing fields, objects, classes or packages that are likely involved in memory anomalies [19, 20, 79].

3.3.2 *Evolution over Time.*

A number of tools also allow to analyze the memory usage evolution of an application over time [24, 25, 78, 118], in its basic form by comparing heap states. In MAT [29], for example, users can compare two heap states by computing a delta type histogram diagram to identify objects with high growth rates. In Dynatrace [28] users can show the number of objects allocated in the selected time interval. Extensively allocated objects can then be considered for reuse, caching, or removal. Other approaches allow to automatically detect growing data structures [117, 119], or to visualize the evolution of the memory composition over time [122, 123].

4 OVERVIEW OF ANTRACKS

The first result of the AntTracks project [115] was a custom Java VM for efficiently collecting detailed memory traces [66]. The AntTracks Analyzer then started as a research prototype for reconstructing heap states from these memory trace files [7]. The tool is now an interactive memory analysis tool for the detection and inspection of various memory anomalies. We selected the AntTracks tool as subject for this study as it is a publicly available¹ and covers more memory analysis tasks than alternative tools. For example, AntTracks can perform detailed analyses over time due to its trace-based nature, while other publicly available tools such as MAT [29] or VisualVM [91] are restricted to snapshot-based (dump-based) analyses. Another reason for selecting AntTracks was the high familiarity of some authors with its code base. The goal of the cognitive walkthrough was to reveal and fix major flaws before the user study, so detailed knowledge of the tool and its implementation was essential.

In the following, we give an overview of a subset of AntTracks Analyzer's features, organized by the memory analysis activities presented in Section 3.

4.1 Memory Growth Detection — Overview

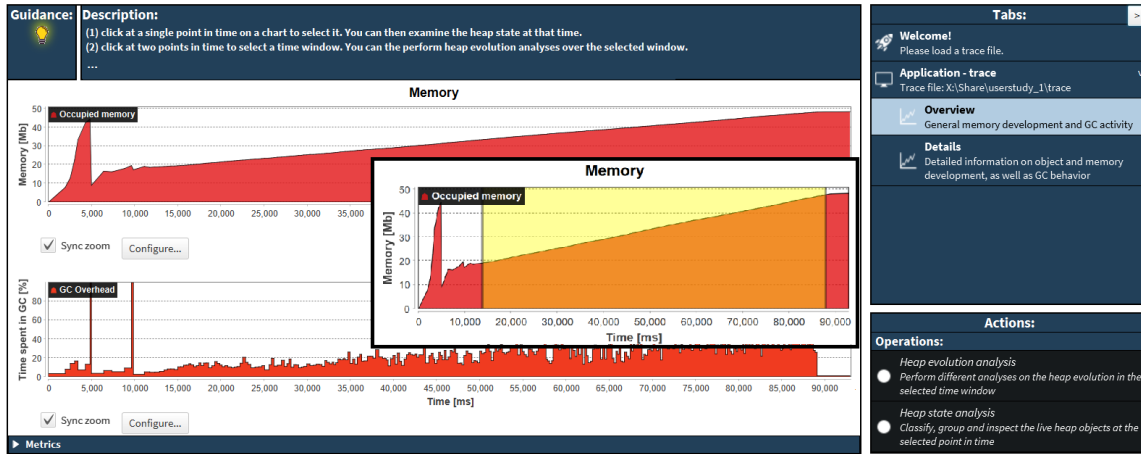
Users working with the AntTracks Analyzer first open a trace file recorded with the AntTracks custom VM. The file contains information on the memory behavior of the monitored application. The *application overview* (see Figure 1a) opens upon loading and shows the memory footprint and GC overhead as time-series charts. A continuous growth of the memory footprint, for instance, may indicate a memory leak. This overview is intentionally kept simple. For example, to avoid terminology unknown to the user, the memory footprint chart only contains a single time series showing the occupied memory. Moreover, it only shows data points marking the end of garbage collections, thereby resulting in a smoother trend line².

4.2 Memory Growth Inspection: Evolution over Time — TrendViz View

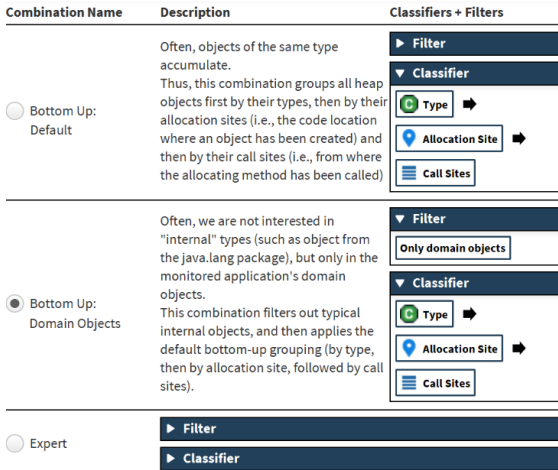
If a user detects a time window with suspicious memory growth, AntTracks' *TrendViz view* [122] allows to inspect the memory evolution during this time window in more detail. The first step is to define properties based on which the heap objects are grouped during analysis (see Figure 1b). For this purpose, AntTracks provides a variety of different object classifiers [121, 124], each of which groups heap objects based on a different criterion. For example, the type classifier groups all objects by their type name, e.g., `java.util.HashMap`. A user can select multiple classifiers for grouping the heap, which results in a classification tree. For example, using the type classifier followed by the allocation site classifier first groups all objects based on their types, and then further groups all objects of a given type based on the source code location they were allocated at. The AntTracks TrendViz visualizes the evolution of the heap based on the selected classifiers (see Figure 1c). When opening the view, a single chart shows only the evolution of the first level of the classification tree, e.g., the evolution of the objects grouped by type. The user can then display further charts for the next levels of the classification tree, e.g., the evolution of the allocation sites of a selected type. For example, in Figure 1c the most-allocated type `Product` has been selected by the user in the top chart (highlighted in yellow), and a second chart below displays this type's allocation sites. This way users can interactively collect information about suspicious objects accumulating over time.

¹AntTracks download link: <http://sww.jku.at/General/Staff/Weninger/AntTracks/Publish/>

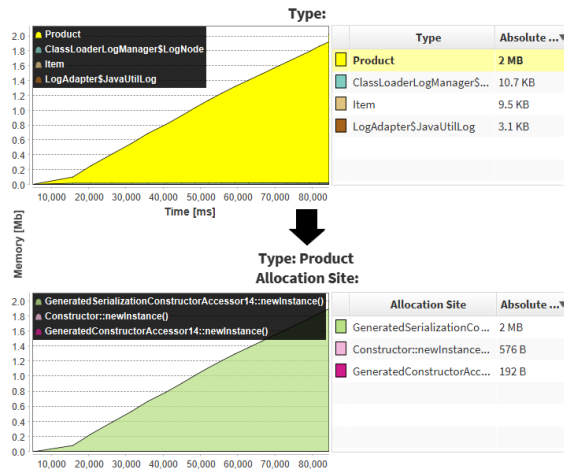
²The occupied memory is generally higher when a garbage collection starts, however, the spikes between garbage collection starts and ends are not relevant for the purpose of detecting memory leaks.



(a) The *Overview* plots the application’s memory footprint and GC overhead and allows to select a suspicious memory leak time window.



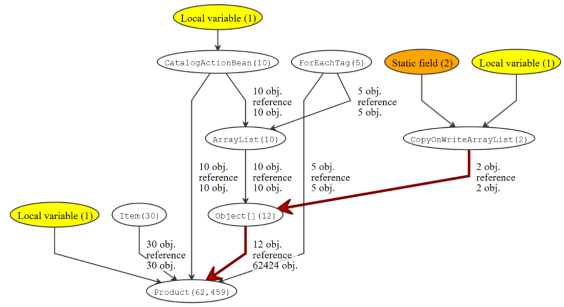
(b) Users can choose from a list of classifier combinations to group heap objects.



(c) The *TrendViz* view displays the heap evolution grouped by the selected classifier combination.

Name	Objects	Shallow size
Overall	879,774	48.2 MB
Filtered	64,074	2 MB
Product	62,577	2 MB
GeneratedSerializationConstructorAcce...	62,553	2 MB
(hidden internal call sites)	~62,553 ...	~2 MB ...
CatalogService::getProduct()	~62,553 ...	~2 MB ...
CatalogActionBean::viewProduct()	~62,553 ...	~2 MB ...
(hidden internal call sites)	~62,553 ...	~2 MB ...

(d) The *heap state* view displays the classified heap at a certain point in time as a tree table.



(e) The *graph* view highlights the paths from a selected group of objects (shown at the bottom) to its most important GC roots (colored nodes).

Fig. 1. Memory leak analysis in AntTracks.

4.3 Memory Growth Inspection: Single Point in Time – Heap State View + Graph View

Users analyzing memory growth over time often reveal suspicious objects that accumulate memory. These objects can then be further inspected at a specific point in time. For example, after a memory growth analysis, AntTracks may suggest to inspect the heap state at the end of the previously selected time window. At this point, all objects that have accumulated during this time window are present in the heap and can thus be easily inspected. AntTracks can visualize the heap state using a *table-based* or *graph-based analysis*.

4.3.1 Table-based Analysis – Heap State View.

When inspecting a specific heap state, the user first selects a classifier combination (cf. [Figure 1b](#)) for grouping the heap objects. The resulting classification tree is displayed in a tree table on the *heap state view*, as shown in [Figure 1d](#). In this table, the user can further inspect suspicious objects previously identified in the trend view. For example, this view allows to inspect the *GC closures* of an object group [116], i.e., the objects kept alive by a certain object group, or a tabular visualization of the path to the closest GC root [116], similar to VisualVM [91].

4.3.2 Graph-based Analysis – Graph View.

Further analyses are needed if a user detects a suspiciously large group of objects being kept alive. This can happen in garbage-collected languages if objects are still directly or indirectly reachable from GC roots such as local variables or static fields. In this case, the user needs to inspect the paths to these GC root to find ways for reducing the number of paths.

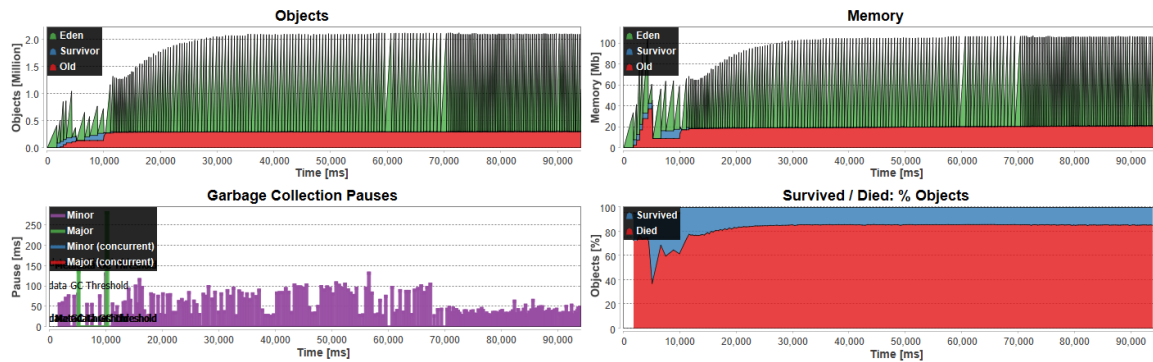
The most convenient way to inspect the paths to GC roots is the *graph view* shown in [Figure 1e](#). Initially, this view only shows a single node representing the set of suspicious objects. By selecting the node the user can apply the *Path to GC roots* operation, which traverses the references pointing to the given objects recursively until GC roots are found. To keep the number of displayed nodes low, objects of the same type are grouped into a single node. Nodes are labeled with their objects' type name and the number of objects belonging to them. Edge labels show how many objects of the top node reference how many objects of the bottom node. GC roots are displayed as special nodes that are highlighted by a colored background. After performing the path to GC roots action, the user can explore the resulting paths and detect the GC roots referencing most objects. To make objects eligible for garbage collections, a developer can then 'cut' the paths to these GC roots by setting references to null or by removing objects from their containing data structures.

4.4 Memory Churn Detection – Details View

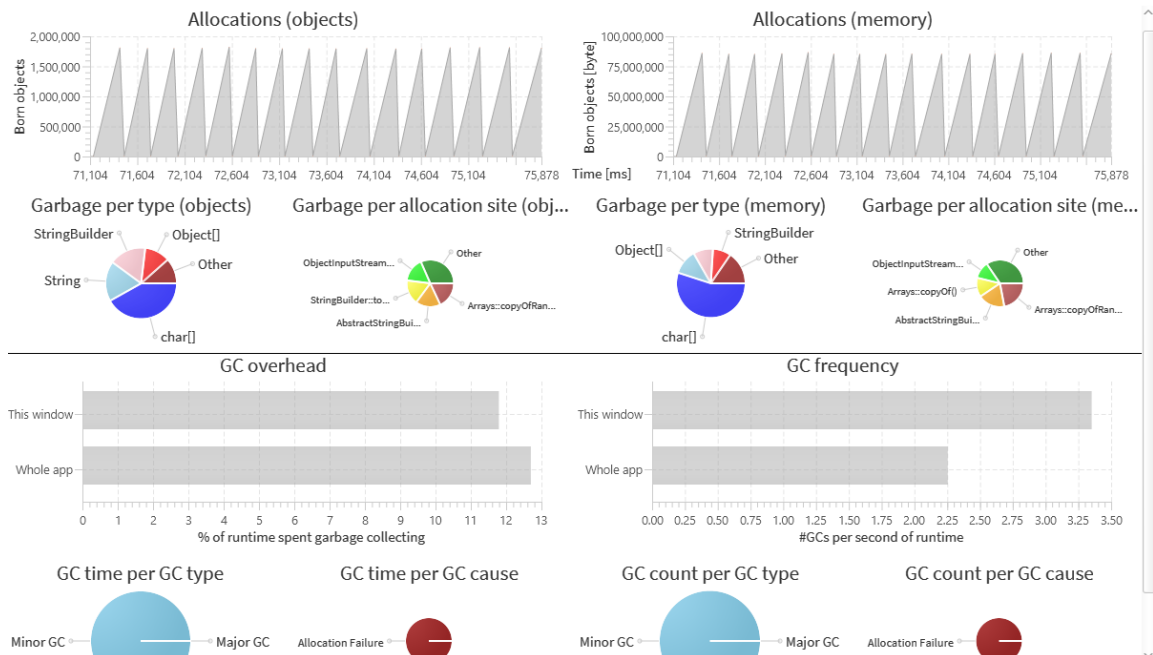
In case of *memory churn* the performance degradation is caused by the creation and garbage collection of many short-living objects [106]. In AntTracks suspicious time windows with high memory churn can be detected in the *details view*, which plots the memory footprint at the beginning and at the end of every garbage collection (cf. [Figure 2a](#)). The memory occupied at the start of a garbage collection is usually much higher than at its end, i.e., the garbage collections appear as spikes. A user aiming to detect high memory churn needs to look for high and frequent spikes in this memory footprint chart.

4.5 Memory Churn Inspection: Evolution over Time – Short-living Objects View

Once a suspicious memory churn time window is detected, the goal of the developer is to reduce the number of allocations. This is achieved by determining the types and allocation sites of the objects responsible for most of the garbage within this window. This information is then used to track down the frequent allocations in the source code to fix the problem.



(a) The *Details* view plots the application’s detailed memory footprint and GC pauses and allows to select a suspicious memory churn time window.



(b) The charts on the *short-living objects* view show the monitored application’s garbage collection behavior.

Filter			
Classifier			
Selected:	Age	Type	Allocation Site
Name	Collected objects	Collected memory	
Overall	54,264,279	2.6 GB	
0 GCs survived	54,102,355	2.6 GB	
char[]	22,640,685	1.4 GB	
Arrays::copyOfRange()	9,694,436	560.4 MB	
(hidden internal call sites)	~9,686,034	~559.4 MB	
CatalogService::getCategoryList()	~9,256,420	~528.1 MB	
CatalogActionBean::viewCategory()	~9,256,420	~528.1 MB	
(hidden internal call sites)	~9,256,420	~528.1 MB	

(c) The tree table on the *short-living objects* view enables users to drill down into object groups causing suspiciously high memory churn.

Fig. 2. Memory churn analysis in AntTracks.

AntTracks detects short-living objects based on the number of garbage collections they survived. This object age information is then visualized in the *short-living objects view* comprising an overview tab presenting various garbage collection metrics to give a first impression of garbage composition (cf. [Figure 2b](#)), and an inspection tab depicting all garbage-collected objects in a tree table using AntTracks' classification mechanism (cf. [Figure 2c](#)). The overview tab helps the user to familiarize with the garbage collector activity in the selected time window. For example, pie charts reveal the object types and allocation sites producing most garbage. By selecting a chart entry the user can switch to the inspection tab to further investigate the respective objects. The inspection tab uses a tree table, similar to the one used on the heap state view, to display the garbage collected objects. Inspecting this tree allows users to determine the objects which did not survive a single garbage collection and to investigate their types, allocation sites, and the methods calling the allocating method.

5 COGNITIVE WALKTHROUGH OF ANTRACKS

We first performed a cognitive walkthrough of the AntTracks Analyzer using the CD framework to assess its usability and to select specific usability aspects for in-depth investigation in the user study. Specifically, three authors of this paper (two of which are familiar with AntTracks' source code) independently assessed AntTracks' different views used during typical memory analysis activities. To do so, every assessor performed a memory leak analysis on the *Dynatrace easyTravel* application [27] and a memory churn analysis on the *http-finagle* benchmark of the Renaissance benchmark suite [93]. Both of these applications have already been used in related work to present typical memory problems [118].

As cognitive dimensions are designed as 'discussion tools' [40], every assessor took notes for each view based on the 14 cognitive dimensions defined in the CD framework [8]. They also rated each cognitive dimension on each view using a color-based three-level classification: (1) green – no issues found, (2) yellow – room for improvement, and (3) red – serious flaws.

After every assessor had independently performed these tasks, their results were merged based on their ratings and their notes during a discussion session. For the rating, they always took the lowest rating of all three assessors as the joint result to ensure all concerns are addressed, i.e., if two assessors rated a CD as green and one rated it as red, the joint rating was red. Each author's notes and comments were merged and discussed to ensure a common understanding.

This merging session resulted in a single spreadsheet shown in [Figure 3](#). The assessor comments are only partially shown due to space constraints. Six cognitive dimensions were regarded as cross-cutting, i.e., affecting the whole application. Overall, 43 view-CD-pairs were rated as yellow (room for improvement) and seven cognitive dimension on two different views were rated as red (serious usability flaws). The serious flaws had to be fixed before the user study to prevent obvious showstoppers during the study.

Due to their high number, further evaluating all of these issues during the user study would have been infeasible. Thus, the assessors jointly selected the 27 most interesting usability issues (highlighted using black font and thick borders in [Figure 3](#)). A list of these cognitive dimensions can also be found in [Table 2](#). This table contains one entry for every view in AntTracks alongside its respective memory analysis activity. For every view, it lists the cognitive dimensions chosen for further investigation during the user study. For each cognitive dimension, we agreed on the methods how data for evaluating the respective cognitive dimension should be collected during the user study: By *observation* (OBS) of user activities and think-aloud statements or by a specific question in the *interview* (INT) at the end of the study. The last column shows the degree of support of a cognitive dimension based on the results of the user study. These results will be discussed in more detail in [Section 7.2](#).

Task	Detection: Memory Growth	Inspection: Evolution over Time	Inspection: Single Point in Time	Inspection: Single Point in Time	Detection: Memory Churn	Inspection: Evolution over Time	Cross-Cutting	
AntTracks View	Overview	TrendViz View	Heap State View	Graph View	Details View	Short-living Objects View		
Cognitive Dimensions	Abstraction	Overview uses easy terminology.	Abstraction into chart series -> improve by ...	Maybe terminology? Data structure DSL ...	Nodes represent groups of objects -> understandable? ...	GC chart	Is the content of the tree view clear? Terminology, icons, etc.	
	Closeness of Mapping	GC chart	Drill-down feature may not be clear. The hierarchical ...	Tree visualized as hierarchical TreeTableView.	How to display different elements (Objects, GC roots ...)	GC chart	Tree visualized as hierarchical TreeTableView.	
	Consistency	Evolution data is by default presented as charts in AntTracks.		Hierarchical data is by default presented as TreeTableView ...	To achieve immersion and closeness of ...		Other column names than on heap state view.	Are there annoying inconsistencies?
	Diffuseness		Overcrowded classifier selection, also see viscosity.	Classifier selection is too complex. Highlight most ...	Test that not too many different notations are used, ...	Explanatory text is too long.	Many charts on overview - too many?	Unnecessary or unnecessarily complex views?
	Error-proneness	Possible flaw: Chart interaction. Positive: Zoom ...		Operations in context menu clear? User-defined ...	Make sure that operations that would create too ...	See Overview (Chart interactions)		
	Hard Mental Operations	Do users recognize growing memory as problem?	See abstraction & closeness of mapping.	User is free to use any classifier combination. Certain ...	Even though users can inspect graphs, the detection of ...	Interpretation of charts hard?	Normal classification trees.	
	Hidden Dependencies	Zoom is synced, selection is synced.	Highlight selection in parent chart better. Also display ...	BUG: New classification in heap state may ...			Link from pie chart to table clear?	Are there any dependencies that we did not find yet?
	Premature Commitment		Time window has to be selected beforehand ...	Time has to be selected beforehand	Time has to be selected beforehand. Once nodes are ...			Order of operations, etc.
	Progressive Evaluation	User can check how many of the suggested time ...	Selected value is shown for every level. The more levels, ...	Position withing classification tree determines progress.	User can always check the path he/she has already ...			
	Provisionality	Can open a new heap state without problems, can ...	All settings can be changed arbitrarily.	Abortion of long running operations is possible.	View is always resettable. Future work: "What-if"-games.			
	Role-expressiveness	Memory chart clear. GC chart probably not directly clear.	Is it clear what a single chart is showing?	Should be clear, ask if the tree table visualization was ...	Are the different types of nodes clear?	Charts maybe not clear, check if users understand what ...	Do users understand the charts?	
	Secondary Notation							
	Viscosity		Inflexibility of the classifier selection. Classifiers cannot ...	Order of classifiers cannot be changed using drag-and-...	Graph grows rather fast.		Order of classifiers cannot be changed using drag-and-...	
	Visibility	New overview tab was implemented: Now Memory + GC ...	Drill-down feature has been improved (with table, etc.) ...	Should be clear, ask if the tree table visualization was ...	Legend was needed.	Many charts at once, may be overwhelming.		Tab system. Do users find out ...

Fig. 3. The spreadsheet documents the results of the CD assessment. Each column represents an activity (cf. row 1) performed on one of AntTracks' views (cf. row 2). Each of the 14 cognitive dimensions [8] is shown in a separate row. A green cell represents a CD for which no issues were found on the respective view, a yellow cell highlights a CD with room for improvement on the respective view, and a red cell depicts serious problems (requiring fixes before the user study). The cell texts show parts of the notes taken by the inspectors during the walkthrough. Highlighted view-CD-pairs (cells with black text and thick border) have been chosen to be evaluated in more detail during the user study.

In the following, we present the assessors' feedback to the cognitive dimensions selected for detailed inspection during the user study. The user study design as well as the interview questions have been adjusted to gain as much insight as possible into these possible usability flaws.

5.1 Memory Growth Detection – Overview

Error-proneness, i.e., the notation invites errors, has been recognized as a likely problematic cognitive dimension in the overview. The most common operations on this view are chart interactions such as the selection of a single point in time (to select a heap state to be analyzed) or the selection of a time window (to select an interval for heap evolution analysis). Various interaction mechanisms (clicking, double-clicking, dragging, etc.) for different actions exist and vary across applications, which can easily lead to misuse. *Hard mental operations* also require attention in the user study. Even though users can be expected to spot continuous growth in a time-series chart, novice users might not relate such patterns to possible memory leaks that should be investigated further.

Table 2. The cognitive dimensions that were chosen based on the results of the cognitive walkthrough to be inspected in more detail during the user study using observations (OBS) and interview questions (INT).

Activity / Capability	Tool Views	Cognitive Dimension	Assessment in User Study	Study Result (cf. Section 7.2)
Detection: Memory Growth	Overview	Error-proneness	OBS + INT	-
		Hard Mental Operations	OBS + INT	o
Inspection: Evolution over Time	AntTracks TrendViz View	Abstraction	OBS	o
		Diffuseness	OBS	+
		Role-Expressiveness	OBS + INT	+
		Viscosity	OBS	+
		Visibility	OBS + INT	o
Inspection: Single Point in Time	Heap State View	Diffuseness	OBS + INT	+/ o
		Hidden Dependencies	OBS	+
		Role-Expressiveness	OBS + INT	o
		Viscosity	OBS	+/ o
Inspection: Single Point in Time	Graph Visualization View	Consistency	OBS	+
		Diffuseness	OBS	+
		Hard Mental Operations	OBS + INT	-
		Role-expressiveness	OBS + INT	-
Detection: Memory Churn	Details View	Error-proneness	OBS	-
		Hard Mental Operations	OBS	-
		Visibility	OBS	+
Inspection: Evolution over Time	Short-living objects View	Consistency	OBS + INT	+
		Diffuseness	OBS	-
		Role-Expressiveness	OBS + INT	o
Cross-Cutting		Abstraction	OBS + INT	-
		Consistency	OBS + INT	+
		Diffuseness	OBS + INT	o
		Hidden Dependencies	OBS	+
		Premature Commitment	OBS + INT	+
		Visibility	OBS	+

5.2 Memory Growth Inspection: Evolution over Time – TrendViz View

Diffuseness, i.e., the verbosity of the notation, and *viscosity*, i.e., the resistance to change, both needed fixing before the user study due to AntTracks' complex classifier system. Although this system is very flexible for expert to arbitrarily group heap objects, this flexibility can make the system difficult to use for novices. In particular, an overwhelmingly large list of available filters and classifiers is presented to the users (diffuseness), who may struggle to select sensible classifier combinations without a solid background in memory analysis. Additionally, the selection and arrangement of these combinations was tedious, for example drag-and-drop features were missing (viscosity). Thus, we extended AntTracks with pre-defined classifier combinations for common tasks before the study (cf. Figure 1b). For example, the combination Bottom-up analysis: Domain objects first applies a filter to omit objects from internal packages (such as `java.lang` or `java.util`), and then groups the remaining objects by their types, followed by their allocation sites and by their call sites. *Visibility*, i.e., the ability to view components easily, was another showstopper CD we fixed

before the user study. The view allows to select a certain object group for drill-down inspection by clicking on its chart series. Yet, the walkthrough revealed that it might not be obvious that an object group can be selected by clicking on the chart. Thus, we added a table next to the chart to make interaction abilities more visible (cf. Figure 1c) and investigated in the study whether users benefit from this additional table. *Abstraction* and *role expressiveness* question if users understand the meaning and the visualization of the drill-down process, i.e., how a classifier combination and the resulting classification tree are represented by multiple drillable subcharts displayed below each other. To emphasize the hierarchical relation between two charts, we added arrows in between charts before the study, as well as a textual description of the drill-down selection, as shown in Figure 1c. Another abstraction we considered to simplify analyses in AntTracks regards the way of presenting allocation sites and method calls. Call chains can become quite long (multiple 10s of calls) and thus hard to inspect, especially if an application employs various libraries that call each other. To reduce the amount of entries in such a call chain, AntTracks creates artificial entries labelled (hidden internal call sites) that combine multiple *internal call sites*, i.e., calls from one method to another inside a packages that cannot be modified by the user (such as `java.util`). This abstraction may be hard to understand for some users.

5.3 Memory Growth Inspection: Single Point in Time – Heap State View + Graph View

5.3.1 Table-based Analysis – Heap State View.

Diffuseness and *viscosity* are also relevant on this view since it uses the same classifier system as the TrendViz view discussed in the previous section. Yet, different classifier combinations are required on both views, as certain combinations are only sensible when analyzing a single heap state but not a trend. We thus improved the system by adding even more pre-defined classifier combinations, but showing only the relevant ones on the respective views. *Role expressiveness* on this view's classifier selection questions whether the different combinations can be distinguished and understood by the users. For each combination, AntTracks shows its name, a description and the list of used filters and classifiers, as shown in Figure 1b. We added an interview question to clarify if this explanation is sufficient for users. *Hidden dependencies* were another problem that became apparent during the cognitive walkthrough. Users can select an object group in a heap state view and then apply various operations, some of which open new windows displaying information related to the selected object group. If the heap state window changes, for example, by selecting a different classifier combination, the object groups on the child windows no longer exist in the parent window, thus breaking (hidden) dependencies. We tried to prevent this problem by opening a new heap state window every time a new classifier combination is applied.

5.3.2 Graph-based Analysis – Graph View.

Consistency is a concern on this view, since its graph-based visualization strongly differs from other AntTracks views, most of which use time-series charts and tree tables to display data. *Role-expressiveness* has to be evaluated regarding the different types of nodes, edges, color encodings and other features that are intended to help users to understand the graph but also pose the risk of being too complex. *Diffuseness* may also be affected by this graph notation. Grouping objects based on their type significantly reduces the number of nodes on the screen but requires additional labeling. This could result in an overly high number of screen objects negatively affecting comprehension by the user. *Hard mental operations* have to be performed as users try to spot suspicious GC roots and suitable cutting points on the paths to them. The view can visualize the heap object graph and roots to the GC roots, yet this information is only useful if users are able to interpret it correctly.

5.4 Memory Churn Detection — Details View

Error-proneness, as in the overview, concerns the interaction with AntTracks' time-series charts, which are the main way of visualization on the details view. *Visibility* asks the question how easily users can detect time windows with high memory churn, i.e., spike-patterns with frequent and tall spikes on the chart, on the details view. *Hard mental operations* are potentially required for users without experience in memory analysis to correctly interpret such spikes as suspicious.

5.5 Memory Churn Inspection: Evolution over Time — Short-living Objects View

Role-expressiveness should be assessed during the study on AntTracks' short-living objects view. Various information (such as garbage composition) is visualized using pie charts, but not all of it may be clear to the users, e.g., due to the terminology used. *Diffuseness* is also of interest as the view contains twelve charts, some of which contain less crucial information and thus could diffuse the more important information. *Consistency* regards the tree table on the inspection tab. In other AntTracks views, the tree table shows the live objects of a certain heap state, while this tree table shows all objects garbage-collected in the chosen time window. We decided to investigate this break in consistency in the study.

5.6 Cross-Cutting Dimensions

Several cognitive dimensions were found to be relevant for all views of AntTracks. We decided to assess in the study if these are well supported. *Visibility* and preventing *hidden dependencies* were our main concerns when choosing a stacked tab arrangement in AntTracks. AntTracks offers a number of different analysis features, many of which open new (child) views. We thus decided on a stacked tab system where each tab can again have further child tabs. *Abstraction* is also important in nearly all tools. For example, typical abstractions are icons or terminology inherent to the given domain. Certain abstractions may be hard to understand in which case they should be fixed in the future. *Consistency* is important for visualizations in tools. AntTracks mostly uses the same chart style on all of its views. Also, much of the information in AntTracks is visualized in tables, often tree tables, since most of its data is arranged in trees (for example classification trees). We included a question regarding consistency in the study questionnaire to reveal potential inconsistencies. *Diffuseness* may concern especially non-expert users. Visualizations should be as clear as possible, and the study was designed to reveal unnecessary or unnecessarily complex parts of the tool.

6 USEFULNESS STUDY DESIGN

Based on the results of the cognitive walkthrough and the subsequent improvements of the tool, our qualitative study assessed the usefulness (i.e., usability and utility) of AntTracks' memory analysis capabilities. We structured our study using the guidelines by Runeson and Höst [101]. For the design of the study tasks, we followed the recommendations by Ko et al. [59]. Specifically, we defined six tasks based on the the initial survey of memory analysis activities and capabilities (cf. Section 3). We iteratively refined these tasks by first testing their difficulty ourselves and then involving a researcher from our lab who had never used AntTracks before as a pilot user. Based on the feedback, we adjusted the study method, e.g., we removed ambiguities in the instructions.

6.1 Study Subjects

We selected the study participants from two sources: (i) we invited students from a course on Java performance monitoring and benchmarking to take part in the study, as the course ensured basic knowledge about the context and purpose of memory analysis tools (cf. Ko et al.'s inclusion condition [59]). Ten students agreed to take part in the study. We made clear in the invitation

Table 3. The study subjects' experience in software engineering and memory analysis.

#	Current Study	Experience in software engineering (in years)	Experience in memory analysis (in years)	Experience in AntTracks
1	Bachelor	3	0.1	No
2	Bachelor	6	0.1	No
3	Bachelor	4	0.1	No
4	Bachelor	3	0.1	No
5	Master	7	0.1	Presentations
6	Bachelor	1	0.1	No
7	Bachelor	2.5	0.1	No
8	PhD	7	2.5	Presentations
9	Bachelor	7	3	No
10	Master	5	0.1	Presentation
11	Bachelor	8	0.5	No
12	PhD	10	0.1	Presentations
13	Bachelor	6	1	Presentations
14	PhD	9	0.5	Presentations
AVG		5.6	0.6	

that the participation and performance in the study are in no way related to the grading of the course. (ii) In addition, four researchers from our department accepted to participate. Table 3 shows a complete list of all participants. None of them had used AntTracks before, yet six of the 14 participant heard about AntTracks in presentations. At the time of the study, nine participants were in their bachelor's studies, two pursued a master, and three were enrolled in a PhD program. Their average experience in software engineering was 5.6 years whereas their average experience with memory analysis tools was 0.6 years. The ten students attending the Java performance monitoring and benchmarking course had completed one homework assignment involving the use of a memory analysis tool such as VisualVM [91] or MAT [29] to inspect the memory behavior of easyTravel [27], a state-of-the-art demo application by Dynatrace [28] that mimics the server of a travel agency. Seven of the students reported this assignment as their only memory analysis experience, which we regarded as an experience of 0.1 years. Besides VisualVM [91] and MAT [29], the participants already had experience in various tools including Java Melody [51, 71], Android Studio [37, 42], Valgrind [81, 111], and Java Mission Control [89] (including Java Flight Recorder [84]).

6.2 Study System

We selected the web application JPetStore 6 [80] as our study system. JPetStore has been widely used in research projects [32, 53, 55, 56, 114]. It models a minimalistic web shop for pets and uses a clearly structured class hierarchy. Categories (e.g., fish) can contain multiple products (e.g., Koi), which in turn can contain multiple items (e.g., spotted Koi and spotless Koi). Categories, products, and items each have their own web page and can be viewed in a web browser. We chose JPetStore since its straightforward structure described in a UML [13] class diagram made it easily comprehensible for the study participants without being familiar with the source code. This helped to mitigate the risk of participants not finishing the study tasks (cf. [120]). To prepare the system for the study, we modified the JPetStore source code to contain the following two memory anomalies.

- (1) *Memory Leak Mode*: In this configuration, we purposely keep objects alive after their intended use. Since memory leaks caused by a single object (e.g., a single static list) can easily be inspected and resolved by a dominator analysis [29], we mimic a more realistic problem that is harder to resolve due to multi-object ownership [116]. To achieve this, every time a product web page is requested, the (normally temporary) Product object shown on the page is stored in and kept alive by two different static lists located in different classes.
- (2) *Memory Churn Mode*: In the original version of JPetStore, displaying a single item results in a database query using the item's ID, causing only a single Item Java heap object to be created. In our modified version, all available items are loaded from the database (a List<Item> of length 10 000) and the needed item is then extracted from this list. This means that 9 999 Item heap objects are needlessly created on every request, a typical case of high memory churn.

We created AntTracks trace files before the user study for both the memory leak and the memory churn mode. In particular, we simulated heavy load of the application by sending numerous requests to the server, requesting the web pages of random categories, products, and items.

6.3 Study Process and Data Collection

We conducted the study in a separate session with each subject. At the beginning of each session, we asked the participants to 'think aloud' [47, 50, 83] during the study. Specifically, we asked them to verbally describe what they were doing, to comment on any of their concerns, and to say whatever comes to their mind while solving the given tasks. A scribe documented the think-aloud statements, while a moderator watched and guided the subjects through the study and took additional notes on interesting observations not covered by the think-aloud protocol. Specifically, we conducted the following process that took approximately one hour per subject:

Preparation. Since the participants worked on the computer of one of the authors, all services and applications that might have distracted a study participant were closed. Before each session, we started the AntTracks Analyzer tool and loaded the trace file that was recorded using JPetStore in memory leak mode. The scribe and moderator prepared their documents to take notes. The moderator additionally prepared a utility questionnaire for the interview at the end of the session.

Briefing. The moderator explained the goals of the study to the participants and asked for their consent on the publishing of the findings [59]. Consent was given by signing a form explaining the study process, the data planned to be collected, and the procedures for storing and processing this data. The moderator also discussed a briefing sheet explaining the JPetStore domain, basic workflows of analyzing a memory leak and high memory churn, and the think-aloud process with the participants. The participants then received a document describing the different tasks to be performed. To ensure focus, the tasks were introduced one after another as the participants progressed through them.

Task Execution. The participants completed the following six tasks:

- *Memory Leak Detection*: They used AntTracks to detect suspicious memory behavior by inspecting the application's memory consumption over time in the Overview tab.
- *Heap Evolution over Time*: The participants selected a suspicious time window and used the TrendViz feature to identify the domain objects showing the highest memory growth in this interval. They further checked at which allocation sites these objects were created, and from which sites they were called.
- *Table-based Heap State Analysis*: The participants opened the heap state at the end of the selected time frame and performed a bottom-up analysis on the domain objects. They identified the objects showing suspicious memory growth in the previous analysis.

- *Graph-based Heap State Analysis*: As a next step, they used the graph visualization to explore the neighbors of these objects. Objects remain alive if they are (indirectly) referenced by GC roots. Participants were asked to follow the from-pointers, in particular, thick edges indicating a large ownership, to find suspicious GC roots. After this analysis, participants used the application’s source code and aimed to fix the memory bug based on their findings.
- *Memory Churn Detection*: After fixing the memory leak, the participants analyzed the second trace file recorded in memory churn mode. They identified suspicious behavior by inspecting the memory charts in the application’s details view.
- *Heap Evolution of Short-living Objects*: Participants were asked to select a time window showing suspicious GC behavior, i.e., frequent collections with high object death rates, for the short-living objects analysis. They were then asked to locate and fix the memory problem in the source code based on their findings.

Data Collection. After finishing all tasks the participants completed a usability questionnaire covering Nielsen’s usability attributes [82] and specific capabilities of AntTracks. The moderator further conducted a semi-structured interview comprising 17 questions on the tool’s utility and usability. The questions are based on the utility questionnaire by Davis [23] and usability issues revealed in the cognitive dimensions assessment (cf. Section 5). More specifically, we included one question per cognitive dimension classified as *Interview* in Table 2, such as ‘Did you experience any problems while selecting a given time or time window?’ (targeting *error-proneness* on the Overview and Details view) or ‘Did you experience any problems with the used terminology, i.e., naming of displayed content and/or icons used?’ (targeting cross-cutting *abstraction*). During these interviews, we also collected the demographic information presented in Table 3. Each interview was concluded with a short debriefing.

Data Analysis and Reporting. After running all sessions we prepared the collected data for analysis. Overall, 370 observations (26.5 per subject on average), 261 think-aloud statements (18.5 per subject on average) and 238 interview statements (17 per subject) were recorded by the scribes, some of which will be quoted in Section 7.2. We labelled all observations, statements and interview answers to allow their systematic use. For example, as shown in Table 4, the think-aloud statement on AntTracks’ overview screen ‘*In the chart, I can see that my memory grows more and more, that is not good.*’ received the labels ‘*Detects Growth In Chart*’ as well as ‘*Recognizes Growth as Problem*’. We adopted and adjusted an iterative labelling process [35] that is similar to Open Coding [104]. First, a set of possible *observation labels* and *statement labels* had to be formed. For this, three of the authors jointly classified a sample set of the study session transcripts. This helped them to gain a mutual agreement on the possible labels and the coding process itself. Then each of the three authors individually coded the remaining observations and statements, while still staying in contact with each other. This allowed the coders to quickly and collectively decide if a new label should be introduced in case an observation or statement could not be mapped to an existing label. In this case, they also went through all previously labelled observations and statements to check if

Table 4. Labeling a think-aloud statement and linking these labels to their relevant cognitive dimensions.

Subj.	Task	Observation / Think-aloud statement	Labels	Relevant cognitive dimensions
1	1	‘In the chart, I can see that my memory grows more and more, that is not good.’	Detects Growth In Chart Recognizes Growth as Problem	Visibility Hard mental operations

the new label should also be applied. Finally, the coders had a joint discussion meeting to merge the three individually labelled lists of observations and statements into a single list, thereby discussing and resolving possible differences.

Each of the labels was then linked to its relevant cognitive dimensions. For example, in Table 4 the two labels are related to the cognitive dimensions *visibility* and *hard mental operations* respectively. Discussing the result of the study (cf. Section 7) can then be done view by view, analyzing each cognitive dimension in question based on the frequency of relevant labels.

7 STUDY RESULTS

We discuss the results regarding the usability of the AntTracks Analyzer memory analysis tool based on the usability questionnaire and the findings for specific memory analysis tasks. We further report findings regarding utility. Recommendations derived from these results are then presented in Section 8.

7.1 Usability Questionnaire

Table 5 depicts the results of the usability questionnaire, which follows Nielsen’s attributes of usability [82]. In the following, we will summarize the result for each of these attributes.

Table 5. Results of the usability questionnaires. We used a four-point scale (0, 1, 2, 3) for *learnability* (very hard to very easy), *error prevention* (too many errors encountered to no errors encountered), *subjective satisfaction* (very bad to very good), and *efficiency* (very inefficient to very efficient). For *memorability*, we used a yes/no question. Aggregations have been performed using the median.

Nielsen’s Attr. / Subj.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Med.
Learnability	3	2.5	2	3	2.5	2	2.5	3	2	2.5	3	2	1.5	2.5	2.5
Overview	3	3	3	3	2	1	2	3	2	2	3	2	2	3	2.5
TrendViz	3	2	2	2	2	2	3	3	2	2	3	2	1	3	2
Heap State	3	3	1	3	3	3	2	3	3	3	3	2	2	1	3
Graph	2	1	0	3	3	0	3	2	2	3	2	1	2	3	2
Details	3	2	3	3	2	2	3	3	3	3	3	3	1	2	3
Short-living Objects	3	3	2	2	3	3	2	2	2	2	2	2	1	2	2
Error Prevention	3	2.5	3	3	3	3	3	2.5	3	3	3	3	3	3	3
Overview	2	2	3	3	2	1	2	2	3	3	3	3	3	3	3
TrendViz	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Heap State	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
Graph	3	2	3	3	3	3	3	3	3	1	3	3	3	3	3
Details View	3	2	3	2	2	2	3	2	3	3	3	3	3	3	3
Short-living Objects	3	3	3	3	3	3	3	2	2	3	3	3	3	3	3
Subj. Satisfaction	3	2.5	2	3	2	2.5	3	3	3	2.5	3	2	2.5	2	2.5
Overview	3	3	2	3	2	2	3	3	3	2	3	3	3	3	3
TrendViz	3	2	1	3	2	2	3	3	3	3	3	2	1	2	2.5
Heap State	3	3	2	3	3	3	3	3	3	3	3	2	3	1	3
Graph	3	1	2	3	3	1	3	3	3	3	3	2	2	2	3
Details View	3	2	2	3	2	3	3	3	3	2	3	3	3	3	3
Short-living Objects	2	3	2	3	2	3	3	2	3	2	2	2	2	2	2
Efficiency	3	3	2	3	3	2	3	3	3	3	3	3	3	3	3
Memorability	3	0	3	3	3	3	3	3	3	3	3	3	3	3	3

Learnability. In general, 13 of 14 subjects found the tool easy or very easy to learn. We see potential for improvements especially with regard to the learnability of the graph visualization, which was twice rated as hard to learn and twice as very hard to learn.

Subject S13, who managed to solve all tasks, surprisingly rated three of the six views used during the study as hard to learn. During the interview, it became apparent that this rating was mainly due to the subject's high familiarity with the MAT tool. MAT offers different views and analysis techniques than AntTracks, which led to some confusion. S3 assessed two of the views as (very) hard to learn, but repeatedly regarded his background in memory analysis as weak.

Error Prevention. Most participants replied that they did not encounter any errors. Participants reporting errors on the overview and the details view struggled with zooming and selecting time windows on the charts. The errors reported for the short-living objects concerned a minor visualization bug.

Subjective Satisfaction. The participants were, generally speaking, very satisfied with the tool. The basic views (overview, heap state table, and details view) had the highest ratings, while the more advanced views (TrendViz view, graph visualization and short-living objects view) were rated slightly lower. These satisfaction scores coarsely match the learnability scores.

An issue that may explain some of these lower ratings was brought up repeatedly during the study and the interviews: AntTracks was mainly perceived as a tool aimed at experienced users. Some novice study participants missed specific guidance that helps them to exploit the tool's full potential. Some users also lacked the background knowledge needed to correctly interpret the visualized data and gain insights from the presented metrics.

Efficiency. We asked all participants if they could productively use the tool in their daily work. Twelve of the 14 participants answered this question with *very efficient* and two with *efficient*. This supports our belief that by further increasing AntTracks' learnability, even novice users could use it efficiently to resolve memory anomalies in their applications.

Memorability. Thirteen of 14 study participants think that they will remember how to work with AntTracks after not using it for some time. Only one participant answered this question negatively.

7.2 Usability Results for Specific Activities

We now discuss the usability of the AntTracks Analyzer in detail by referring to the think-aloud statements (THA), observations (OBS), and the answers to the interview questions (INT) obtained during the study. As discussed in [Section 5](#), we focus on a number of cognitive dimensions per view during this analysis. For each selected cognitive dimension we provide a *study result* (listed in [Table 2](#)), a statement summarizing the results, as well as a more detailed study report.

7.2.1 Memory Growth Detection — Overview.

The cognitive dimensions we studied in more detail on this view are the *hard mental operations* possibly needed to detect and interpret suspicious memory growth and the *error-proneness* of interacting with the time-series charts.

Although all participants could detect the memory growth, only 70% of them managed to select a good time window for analysis. Thirteen out of 14 participant mentioned that the application contains suspicious memory growth (THA). Similarly, during the interview, all 14 participants stated that they had no problem to detect the memory growth (INT). Nevertheless, we observed that this did not mean that users could also select a good window for analysis. As shown in [Figure 1a](#), the memory chart shows a tall memory consumption spike during initialization – a typical memory pattern that is not related to a memory leak. Still, four of the 14 participants

selected a time window covering the spike instead of the memory growth for further analysis (OBS), while two other participants expressed (THA) that they probably should investigate the spike in addition to the continuous memory growth. After the moderator explained that a time window covering the memory growth should be selected, over half of the participants expressed concerns regarding the optimal size of the time window (THA). (*Hard mental operations: Medium*)

Unintuitive controls caused problems with chart interaction. We observed that AntTracks' chart interactions confused most of the participants. The text next to the chart explaining the different ways of interaction was often ignored (OBS). Overall, only three out of 14 participants had no problems when interacting with the chart (INT). Rather, typical statements were 'The controls are not clear and the explanation text is too long' (THA - S6), 'How do I select a time window?' (THA - S1, S5), and 'Intuitively, I would have dragged for time window selection' (INT - S4). Many participants rated this fact as an error, which is also reflected in the usability questionnaire (cf. Table 5). (*Error-proneness: Bad*)

The position of tabs and buttons led to irritations. Twelve out of 14 participants struggled to find the button to start the heap evolution analysis (OBS). The AntTracks version used in the study displayed the list of all open tabs and the list of available operations on the left side of the screen. Five people suggested to place the list of tabs and operations on the right side of the screen (INT) (as shown in Figure 1a), since 'It is typical to look for buttons in the bottom-right corner' (INT - S5). (*Other findings - Visibility*)

7.2.2 Memory Growth Inspection: Evolution over Time — TrendViz View.

We studied five cognitive dimensions on the TrendViz view as shown in Table 2.

AntTracks default classifier combinations allowed novices to select suitable classifiers for heap object grouping. AntTracks' classifier selection system has been reworked before the user study to provide default classifier combinations to choose from, hiding its complex selection dialog in an expert mode. Except for two participants, one of which wanted to try the expert mode out of curiosity (OBS - S5, S13), all other participants selected the correct classifiers without problems by using one of the pre-defined classifier combinations. (*Diffuseness: Good; Viscosity: Good*)

Participants succeeded with finding the drill-down feature. We added interactive tables next to the charts before the study, to allow the users to drill-down by clicking on table entries. This paid off, as all participants except one discovered the drill-down feature (OBS). During the interview, we asked the participants whether they were aware of the drill-down feature in the view. A typical response was 'No. But I intuitively clicked on the table and expected something to happen.' (INT - S1, S7, S9, S10, S11). One participant stated that he 'would probably have clicked on the chart if it contained a hover effect combined with a changed mouse cursor and a tool tip' (INT - S4). (*Visibility: Medium*)

Participants understood the drill-down feature and sub-chart dependencies but were confused by terminology. Eleven subjects had no problem understanding the drill-down feature (INT), two subjects said it took them some time (INT - S6, S13), and one participant had general problems to understand the TrendViz view (OBS - S3). When asked whether they understand what happens when they drill-down, answers included 'More details of the selected elements are shown' (INT - S2), 'It is like following one branch in a tree' (INT - S4), and 'Selecting a type to show its allocation sites was clear to me' (INT - S1). Thus, we can conclude that in general the meaning of the drill-down was clear to the subjects. Yet, we observed that four subjects stopped the drill-down early due to bad abstraction. Instead of investigating all available four call sites (by performing four drill-downs) they stopped at the first call site, i.e., '(hidden internal call sites)' (OBS), as they expected no additional call sites (THA). (*Abstraction: Medium; Role-expressiveness: Good*)

7.2.3 Memory Growth Inspection: Single Point in Time — Heap State View.

The cognitive dimensions *viscosity*, *diffuseness*, and *role-expressiveness* on this view are relevant for the classifier selection, as well as for the view in general. The problem of broken *hidden dependencies* was no longer an issue after fixing a potential problem before the study (OBS).

Half of the participants were unaware of the direct switch from heap evolution analysis to heap state analysis. Even though the TrendViz view offers easy switching to the heap state analysis at the end of the selected time window, seven participants were unaware of that feature and instead used a complex work around, i.e., going back to the overview screen and manually selecting this point in time on the chart (OBS). (*Viscosity: Medium – Navigation between views*)

The participants effectively used the pre-defined classifier combinations for typical heap state analysis tasks and selected them based on either name or description. In the interview we asked the participants about the selection of the classifier combination in the heap state view. Seven said that they selected the correct classifier combination based on its *name* while six answered that they used the textual *description* for selection (INT). Three participants stated that they additionally checked the set of classifiers to make sure it matches the description (THA). Only a single participant selected a wrong combination by ignoring all information and by just using the pre-selected default combination (OBS - S10). (*Diffuseness: Good; Viscosity: Good – Classifier Selection*)

Some participants regarded the tree table view as too diffuse. While inspecting the heap state, three participants mentioned that ‘*too much information is shown on the tree table view*’ (THA - S3, S5, S14), and that they wished for a master-detail view (INT) to reduce the view’s *diffuseness*. (*Diffuseness: Medium – Heap state table*)

Hiding the complexity of the classifier system made selection easier but half of participants no longer understood its full possibilities. In the interview, half of the participants said that it was obvious how the classification system works and that the pre-defined combinations just differ in terms of which filters and classifiers are applied (INT). Four said that they were aware of some classification system, understood the results intuitively, but did not think about how it works in the background (INT). Three others said that they did not recognize a classifier system as such but just selected a classifier combination name based on the instructions (INT). (*Role-expressiveness: Medium*)

Participants struggled with terminology. We again observed problems regarding the cognitive dimension *abstraction*, as some people struggled with terminology used on the heap state view. For example, two participants did not understand the meaning of ‘*hidden internal call site*’ (THA - S5, S10), while one could not make sense of the term ‘*retained size*’ (THA - S6). (*Other findings – Abstraction*)

7.2.4 Memory Growth Inspection: Single Point in Time — Graph View.

The object graph visualization in AntTracks enables users to visually explore the paths from a group of objects to their GC roots. We received valuable feedback during the study regarding this view.

Meaning of graph view elements not immediately clear. Only five subjects said that all operations and elements on the view were clear (INT). Four subjects said that they need better edge labeling (INT) to understand how many objects are kept alive on which path (THA, INT) and three subjects suggested to explain the operations in more detail, e.g., by using a tool tip (INT). One subject suggested to reduce the amount of text used in GC root nodes and to highlight different GC root types in different colors (INT - S8). Two participants generally struggled with using this view (OBS - S3, S6). For example, one participant did not understand that nodes and edges represent

groups of objects and object references respectively but instead thought the graph would visualize relationships between outer and inner classes (THA - S6). (*Role-expressiveness: Bad*)

Graph complexity regarded manageable. Even though the graph view displays many elements, no participant expressed any concerns, possibly also due to AntTracks' highlighting of important edges with thick lines to guide the users. This is further confirmed by the fact that half of the participants liked this edge highlighting (THA, INT). (*Diffuseness: Good*)

Interaction in graph-based visualization worked fine. As this is the only view in AntTracks that does not use tables or time-series charts for visualization, we studied whether participants have problems with this break in consistency. However, all subjects managed to apply the operations and use the features for zooming and panning, and no problems with the interaction mechanisms were found (OBS, THA). (*Consistency: Good*)

Half of the participants struggled to extract the information needed to fix the memory leak. Finding the most suspicious GC roots relies on detecting those paths on which the most objects are kept alive. However, as already mentioned, edge labeling was a source of confusion (OBS, THA). Also, two subjects (correctly) expressed concerns whether thread-local variables (that were at that time visualized in the same way as the more dangerous static fields) should also be considered as suspects (THA). In addition, we observed that three participants ignored the fact that the memory leak was caused by multi-object ownership, i.e., two different static fields kept the same objects alive, even though they were both shown side by side on the graph view (OBS). (*Hard mental operations: Bad*)

Two thirds of the participants managed to trace the memory leak to source code. Overall, six of the 14 participants were able to identify both static lists that are responsible for the memory leak within the source code (OBS). Three participants found one of the two sources for the memory leak, but did not recognize the multi-object ownership in the graph view (OBS). Five participants were not able to use the insights gained in the tool to make the necessary fixes in the source code (OBS). (*Other findings*)

7.2.5 Memory Churn Detection — Details View.

The second part of the study involved the analysis of memory churn. The participants' first task was to inspect the development of the application's memory footprint over time and to look for memory anomalies using AntTracks' details view. Since this view uses the same charting technique as the overview, the problems regarding the *error-proneness* of chart interactions reported in [Section 7.2.1](#) also affected this view.

Novices struggled with correctly interpreting the spike pattern. The main goal on the details view is to detect memory churn hotspots based on spike patterns on the memory chart, as shown on [Figure 2a](#). Twelve of 14 participants recognized this spike pattern (THA). Two participants wrongly focused on the initial memory spike as discussed in [Section 7.2.1](#) (OBS - S1, S3). Five participants had no idea whether such spike patterns are abnormal and have to be inspected further (THA). The statements ranged from '*I think many short-living objects are normal in Java applications*' (THA - S12, S13) to '*It cannot be normal to allocate two million objects when only showing a few webpages*' (THA - S11). Selecting a good time window for the analysis was also hard for most participants as the size of the selected time windows varied drastically from 2 to 200 garbage collections (OBS). Very short or very long time windows may distort the analysis result, and a very large time window may unnecessarily increase the analysis time. (*Visibility: Good; Hard mental operations: Bad*)

GC-specific terminology confused some participants. The details view shows three different series per memory chart, representing the different generations of the GC: *Eden*, i.e., objects that are new and have not yet survived a single garbage collection, *Survivor*, i.e., objects that have

survived a few garbage collections, and *Old*, i.e., objects that have lived for a long time and are stored separately in the heap in order to speed up frequent garbage collections. Seven participants stated that these terms are either unknown to them or that they once learned about them but have forgotten them (THA, INT). (*Other findings*)

7.2.6 Memory Churn Inspection: Evolution over Time — Short-living Objects View.

The upper half of the short-living objects view presents pie charts showing the types and allocation sites that caused the most garbage in terms of objects and bytes in the selected time window (cf. Figure 2b). The bottom half of the view shows charts comparing the selected time window's GC frequency and GC overhead with the application's average. During the study, we studied the *role-expressiveness* and *diffuseness* of these charts. Additionally, the view provides a tree table view for detailed analysis which we inspected regarding *consistency*.

All participants managed to extract the information needed to inspect the problem despite some irrelevant and overly complex charts. All subjects managed to use the charts to recognize the type that caused the most garbage (OBS). The allocation site pie charts were, in general, understood but ignored (OBS, THA). Only one participant said that the allocation site charts are unclear (INT - S13). Yet, four participants expressed that they did not understand the charts on the bottom half of the screen (INT), while six other participants ignored these charts because they considered them as not relevant to the problem (INT). (*Role-expressiveness: Medium; Diffuseness: Bad*)

The participants had no problems with interpreting the tree table. After inspecting the charts, the participants switched to the tree table to inspect the garbage created over the selected time window. Since tree tables in AntTracks are generally used to visualize *live objects*, we wanted to know whether the study participants understood that *dead objects* are shown in this case. Thirteen participants stated that the content of the tree table was clear to them (INT). Only one participant replied that he '*had to think a bit about what the columns mean*' (INT - S8). (*Consistency: Good*)

Most participants could locate the memory churn location in the source code, six could to fix the problem. While looking for the root cause of the memory churn, all subjects started to inspect the top-most table entry, i.e., char arrays that died without surviving a single garbage collection (OBS). Since domain types (e.g., types that are not part of the Java library) are often easier to locate in the source code, AntTracks highlights these types in bold to direct the users' attention to these tree table entries (for example, in Figure 1d the type *Product* is highlighted as a domain type). Still, only half of the participants noticed the domain object type that generated the most garbage (OBS). During the interview, one participant stated that he would '*probably have investigated the domain objects if they were not only written in bold, but shown in a separate section within the tree table, or even as a separate pie chart on the overview*' (INT - S3).

Overall, eleven participants found the problematic method in the source code: all seven participants who inspected the domain objects in AntTracks and four of the seven subjects who did not inspect the domain objects in AntTracks. Only six of the eleven subjects who managed to locate the problematic method were also able to fix the problem. Five of them had inspected the domain objects (OBS). Subjects who did not focus on the domain objects in AntTracks were looking for allocations of char arrays (THA), while the others were looking for *Item* objects (THA). Many *Item* objects are created by the problematic code, but since every *Item* object contains up to eight *String* objects each again containing a char array, all these allocations are represented by the same call site, which obviously confused some of the subjects (OBS). (*Other findings*)

7.2.7 CD Assessment: Cross Cutting.

Several cognitive dimensions are relevant on all views or relate to the general structure and organization of AntTracks.

Constraints on the order of doing things. Participants did not feel constrained by the tool, which allows to go back after making a wrong step and to revert earlier actions (OBS). (*Premature commitment: Good*)

Basic layout and stacked tabs. We did not detect any hidden dependencies the participants were not aware of (OBS). When asked what they liked about the tool in general, six participants said that they like the general layout using stacked tabs and the consistent positioning of tabs and actions on the screen (INT). (*Visibility: Good; Hidden dependencies: Good*)

Assumptions made regarding terminology knowledge. We noticed that especially novice users occasionally struggled with terminology (OBS). For instance, half of the participants did not know the difference between eden, survivor and old objects on the details view (INT). Two participants did not understand the difference between an allocation site and a call site (INT - S7, S12) – a problem also affecting other participants as we suspect based on our observations (OBS). (*Abstraction: Bad*)

Consistency of visualizations. The participants did not point out inconsistencies regarding different means of visualization in AntTracks (INT). AntTracks' chart syncing, i.e., keeping the same zoom levels and time selections across multiple charts and views, was positively mentioned with regard to consistency (INT - S10). (*Consistency: Good*)

Suggestions for reducing complexity. We interviewed the study participants whether they considered any of AntTracks' views or interface elements as unnecessary or overly complex. Eight participants raised no concerns in this regard (INT). The suggestions for reducing complexity encompass a master-detail view on the heap state view (INT - S3, S14), the simplification and improved guidance on the graph visualization (INT - S6, S8, S12), and the removal of some charts on the short-living objects view (INT). On the positive side, two users commended AntTracks' good default settings while keeping expert modes for advanced users (INT - S4, S9). (*Diffuseness: Bad*)

7.3 Utility

To assess the utility of AntTracks' views, we first asked subjects whether they liked the flow suggested by the study, i.e., to first search for memory growth, then visualize the growth using the AntTracks TrendViz, and then inspect the final heap state followed by an analysis of the heap object graph. Twelve subjects found the order natural (INT), while two participants stated that they did not really need the trend analysis (INT - S2, S8). However, we also received positive feedback for the trend analysis, for example *'In hindsight, I liked the trend view. Without the study tasks, I would have directly selected a single point in time because I am used to it'* (INT - S4).

We also asked the participants whether they missed a specific feature or whether they wanted to make any other comments, resulting in a catalog of 19 feature requests and possible improvements for AntTracks. The most-wished feature mentioned by five study participants concerns *'better guidance'* (INT). This is related to three sub-problems: (1) deriving knowledge from the current screen, i.e., interpreting the shown data; (2) selecting appropriate steps which should be taken next based on the findings on the current view; and (3) guidance within the IDE on how to fix problems in the source code. Three participants wished for *'more tool tips'* across the various views, especially on the graph view (INT). Some improvements were suggested by at least two participants, such as a *'master-detail view'* for views using tree tables (INT - S3, S5), as well as tutorial videos explaining how to use the tool (INT - S3, S6). These videos could be uploaded to a video platform (such as Youtube), as well as directly integrated into AntTracks for ad-hoc learning / learning by doing [103] (INT). All other suggested improvements were suggested by a single subject and describe rather minor changes, often with potentially high impact, e.g., zooming with CTRL-key and mouse wheel (INT - S5) or highlighting the currently hovered line in the tree views for better orientation (INT - S8). There was also a wish for more automated analyses (INT).

Finally, we asked the study participants what they liked and disliked about the tool in general. Most participants mentioned that they liked AntTracks' general look-and-feel as well as its layout and visual structure (INT). Four participants said that they liked the charts, i.e., their structure, labeling and the selection synchronization feature (INT). Two participants mentioned AntTracks' high productiveness (INT) and also found it intuitive to use (INT). The most disliked aspect of the tool was the chart interaction. Five participants mentioned that they did not like the way how zooming and time window selection works (INT), two participants pointed out that AntTracks and especially its charts react laggy to user input (INT). Four participants said that, although they liked the way how the tabs and operations are arranged, they should be placed on the right side of the window since this is where most users look for operation buttons (INT). As already mentioned it earlier, three participants repeated that they lacked guidance for novice users (INT).

8 RECOMMENDATIONS

Based on the study results presented in the previous section we derived nine general recommendations for developers of memory analysis tools. Table 6 shows each of these recommendations as well as cross references to the study results based on which we formulated the recommendation. In the following, we shortly describe each of them.

Table 6. Summary of recommendations

Name of recommendations	Cross references to relevant study results
Use flexible drill-down mechanisms	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Graph view (Section 7.2.4), Short-living objects view (Section 7.2.6)
Hide complexity using task-specific default settings	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Cross cutting (Section 7.2.7)
Carefully select and explain memory analysis terminology	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Details view (Section 7.2.5), Cross cutting (Section 7.2.7)
Show details and advanced analysis results only on demand	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Short-living objects view (Section 7.2.6), Cross cutting (Section 7.2.7)
Provide support for time selection in large time series	Overview (Section 7.2.1), Details view (Section 7.2.5)
Ensure a smooth transition from evolution analysis to snapshot analysis	Heap state view (Section 7.2.3)
Use automation to relieve users from complex tasks	Overview (Section 7.2.1), Details view (Section 7.2.5), Graph view (Section 7.2.4)
Provide guidance and explanations to support exploratory learning of analysis capabilities	Overview (Section 7.2.1), Graph view (Section 7.2.4), Details view (Section 7.2.5), Cross cutting (Section 7.2.7)
Provide IDE integration to guide diagnosis of memory bugs	Graph view (Section 7.2.4), Short-living objects view (Section 7.2.6)

Use flexible drill-down mechanisms. AntTracks heavily relies on its classification system with multi-level grouping [121, 124]. During the study, we were able to observe that though not all participants recognized the classification system as such even novice users were able to easily understand the resulting tree-structured data. Allowing the users to drill down on this data level by level (e.g., from the whole heap to the most allocation intensive type further to this type's most allocation intensive allocation site) enables them to focus on one thing at a time. The same could be observed on the graph view. The participants performed best when they could focus on one task, inspecting one path to a GC root at a time, drilling down step by step.

Hide complexity using task-specific default settings. Even though this may sound trivial, we can clearly see that the study participants profited from pre-defined classifier combinations for different typical analysis task. Before introducing these pre-defined combinations, AntTracks also pre-selected a default combination, yet this combination had to be slightly adjusted based on the task's analysis goal - a seemingly simple task which nevertheless can demand too much from novice users. Thus, we suggest to define typical tasks and workflows and provide default settings for each of them.

Carefully select and explain memory analysis terminology. Probably one of the most clear results of the study is that AntTracks used too much terminology that is unknown to novice users (such as *allocation site*, *call site*, *retained size* or the garbage collector's space names *eden*, *survivor*, and *old*). Tool developers should aim to improve this situation by reducing the number of complex terms in situations where they are not really needed and by providing easy-to-understand explanations.

Show details and advanced analysis results only on demand. As stated above, the participants performed best when they were able to focus on a single task, drilling down on the problem step by step. This also relates to the well-known visual information-seeking mantra by Shneiderman: '*Overview First, Zoom and Filter, Details on Demand*' [105]. By first giving an overview and only showing details on demand, the view's diffuseness is reduced and its visibility is increased. This does not only concern the analysis views, but also complex configuration views or settings panes. For example, AntTracks by default hides the expert mode on the classifier selection view, yet offers very flexible configuration possibilities on demand. Other examples in AntTracks encompasses the *Overview* screen, which shows only two charts, both of which only contain a single chart series, to be as easy to understand as possible. For more detailed information, the user can switch to the *details view*. A counter-example is AntTracks' *short-living objects view*, which contained far too many charts. Since many of these charts are not needed to find the root cause of high memory churn but rather present general and additional information on the GC activity, they should be moved to another tab to reduce diffuseness.

Provide support for time selection in large time series. This recommendation is two-fold: First, it suggests to use well-known and established user input handling on time-series charts, and second it suggest to provide tool support for intelligent (semi-)automatic selections. For example, regarding the first recommendation, study participants suggested to use input handling similar to Audacity³. This application provides a wide array of interaction possibilities such as the selection of time windows by dragging the mouse, where these time windows can be further modified, for example moved or resized. Regarding the second recommendation, by analyzing the chart's underlying time series data [34], the tool can automatically detect suspicious patterns (such as continuous memory growth or frequent memory spikes). When such a pattern is detected, the tool can then suggest a

³Audacity [3, 70] is a free, open source, cross-platform audio software.

time windows for memory analysis inspection to the user [118], further reducing the need for hard mental operations.

Ensure a smooth transition from evolution analysis to snapshot analysis. In general, we can distinguish two types of analyses that can be performed in memory analysis tools: Analyses that inspect the application at a given point in time, and analyses that inspect the application's behavior over time in a given time window. Both of these analysis types are often interwoven and follow each other in a typical analysis workflow. For example, when investigating memory leaks, the user typically first inspects which objects accumulate *over time* in a certain window, and then inspects the object graph around these objects at the end of the time window, i.e., at a given *point in time*, to find the reason for their accumulation. By defining typical workflows, i.e., typical orders of analysis steps, users can be better guided through the whole process, making transitions between different types of analysis easier to follow.

Use automation to relieve users from complex tasks. This recommendation mainly focuses on reducing the amount of hard mental operations. During the study, especially on the graph view, the participants expressed desire for automation to reduce the amount of manual work needed during the analysis. For example, they suggested that paths to the GC roots could automatically be calculated and analyzed, only showing those most likely involved in a potential memory leak. The previously mentioned automatic suggestion of suspicious time windows also relates to this recommendation.

Provide guidance and explanations to support exploratory learning of analysis capabilities. Another main finding of the study is that novice users, even though often able to 'see' suspicious patterns, cannot correctly interpret them and thus cannot derive the right conclusions. This problem already became apparent during the first task of the study, in which the users had to inspect a memory time-series chart and select the most suspicious time window. While all of the participant saw the continuous memory growth there, 30% of the participants chose another time window. Also, one of the first questions by a participant was '*Is there some sort of suggestion available?*'. Similar interpretation problems could be observed throughout the study. We thus suggest that tools should not only provide the functionality to inspect memory anomalies, but they also need to provide guidance [33] to support exploratory learning and learning-by-doing [103] and to increase the tool's general learnability [1, 60, 82]. Such guidances exist in various forms, ranging from simple wizards [26, 110] to context-sensitive help systems such as coaches, guides or advisors [26, 74]. For example, advisors are context-sensitive help systems that usually include hints, tips, reasoning support, and explanations of complicated concepts, helping novice users to make decisions and helping to understand why a certain step must be performed or to determine why a certain decision was suggested. Such help systems, in combination with more recent approaches such as micro-learning and gamification [38, 43, 49], are often used during the process of onboarding, i.e., introducing a new tool or service to a person to ensure success [98, 107].

Provide IDE integration to guide diagnosis of memory bugs. Finally, we suggest a stronger integration of memory analysis tools with IDEs. Currently, most memory analysis tools are standalone applications, decoupled from the user's development environment. Yet, it is this development environment where the user is expected to fix the just detected memory anomaly, however, without further guidance. Developing an IDE plugin [4, 21] (and probably even expanding the tool's capabilities by the use of hybrid static and dynamic analysis [31]) would make it possible to automatically detect and highlight suspicious code segments and provide further guidance on the source code level.

9 PLANNED IMPROVEMENTS FOR ANTRACKS BASED ON RECOMMENDATIONS

The previous section distilled a set of recommendations based on the results of our study. We now report our improvement plans for five areas of AntTracks based on these recommendations.

9.1 Allocation Site Handling

In complex software systems, call chains can become confusingly long. To mitigate this problem, AntTracks already distinguishes *domain call sites* likely part of the application under investigation and *non-domain call sites* in libraries, for example classes located in packages `java.*` or `javax.*`. Call chains stretching over multiple non-domain call sites are collapsed into a single entry named *hidden internal call sites*. In JPetStore, the test system inspected in the study, this often shortens call chains containing more than 25 entries down to five entries. Unfortunately, we observed that some participants still struggled with interpreting this entry and they also had problems to distinguish the terms allocation site and call sites. This means that the presentation of allocation sites and their call chains has to be further improved as expressed by our recommendation to *carefully select and explain memory analysis terminology*. Non-domain call sites could be hidden completely by default, only displaying them on demand, thereby implementing our recommendation to *show details and advanced analysis results only on demand*. For domain call sites, the call distance could be shown next to them, i.e., either *directly* called by X or *indirectly* called by X. Another planned feature is to show the call chain not textually but visually using a graph. Such a call graph could also be combined with techniques for hiding internal call sites (discussed above). Finally, a stronger coupling between the analysis tool and the IDE would realize our recommendation to *provide IDE integration to guide diagnosis of memory bugs*. Selecting an allocation site in the analysis tool could highlight the source code location in the IDE. We expect that this will help users to interpret the call hierarchy more easily as compared to just looking at the allocation site information in the analysis tool.

9.2 Domain Filtering

The idea of distinguishing non-domain call sites and domain call sites can also be applied to types, thereby distinguishing non-domain types and domain types. Our study showed that participants focusing on domain call sites and domain types were more successful in resolving the underlying problem in the source code. In AntTracks, domain sites and domain types are currently written in bold to gain the users' attention. Another technique to highlight domain objects is to automatically group non-domain objects and domain objects, thereby implementing our recommendation to *use automation to relieve users from complex tasks*. However, not every memory anomaly may be resolved by just looking at domain objects. For example, applications that mostly work with built-in data types may not profit from such a feature. Nevertheless, following our recommendation to *hide complexity using task-specific default settings* we think that inspecting the domain objects first reduces task complexity, in many cases already provides useful results, and sometimes even reveals the root cause of the underlying problem.

9.3 Chart Interaction Behavior

Many study participants were dissatisfied with the way AntTracks handles the zooming and the selection of time windows on time-series charts. The decision to use this convention was partially based on the default zooming behavior of the JFreeChart charting library [36], which defines zooming as mouse drag. We will thus improve AntTracks' chart interactions, thereby implementing our recommendation to *provide support for time selection in large time series*. To this end, we will

study other tools and applications with similar chart interaction techniques (such as Audacity [3]) and take them as example.

9.4 Graph-based Views

Most state-of-the-art tools rely heavily on the visualization of data using (tree) tables. Yet, ample scientific work exists on more advanced features for memory visualization [2, 46, 73, 76, 97, 102, 131]. AntTracks thus already provides a graph-based visualization of the aggregated object graph to inspect the paths to the GC roots. The study revealed interesting findings in this regard: Firstly, GC roots were visualized in the same way regardless of their kind. However, thread-local variables are often very short-living and thus in most cases not relevant for identifying memory leaks. Visualizing such roots in the same way as more suspicious roots such as long-living static fields can distract tool users. We thus plan to use different notations to better highlight more suspicious GC roots. Secondly, the edge labels currently show how many objects from a given edge’s start node reference objects of the given edge’s target node. This makes it easy to inspect the ownership relationship between two neighboring nodes, but it remains hard to extract the ownership influence between two more distant nodes. For example, in Figure 4a, we see a part of the graph visualization the participants were confronted with during the study. It shows that nearly all Product objects (bottom-most node) are referenced by twelve different Object[] objects, two of which are referenced by CopyOnWriteArrayList objects and ten are referenced by ArrayList objects. Taking only edge labeling into account, this notation is not helpful to understand if the CopyOnWriteArrayList or the ArrayList objects keep alive more Product objects. To this end, we also encoded the ownership, i.e., how many objects are kept alive by a certain node, into the edge color and width. Even though the thicker colored edges guided the participant into the right direction, some reported that they did not understand why certain edges were colored and wider than others. This suggests that edge labeling and edge highlighting in memory object graphs should not be too general but problem-driven, as suggested by our recommendation *hide complexity using task-specific default settings*. For example, we meanwhile already realized a new graph view in AntTracks supporting the visualization of ownership edges, see Figure 4b. Instead of labeling edges based on the relation between two neighboring nodes, it takes one node, i.e., object group, and adjusts all edges and their labels to highlight those that keep alive most of the selected objects. The labels in Figure 4b now clearly highlight the path on which most Product objects are owned, i.e., kept alive.

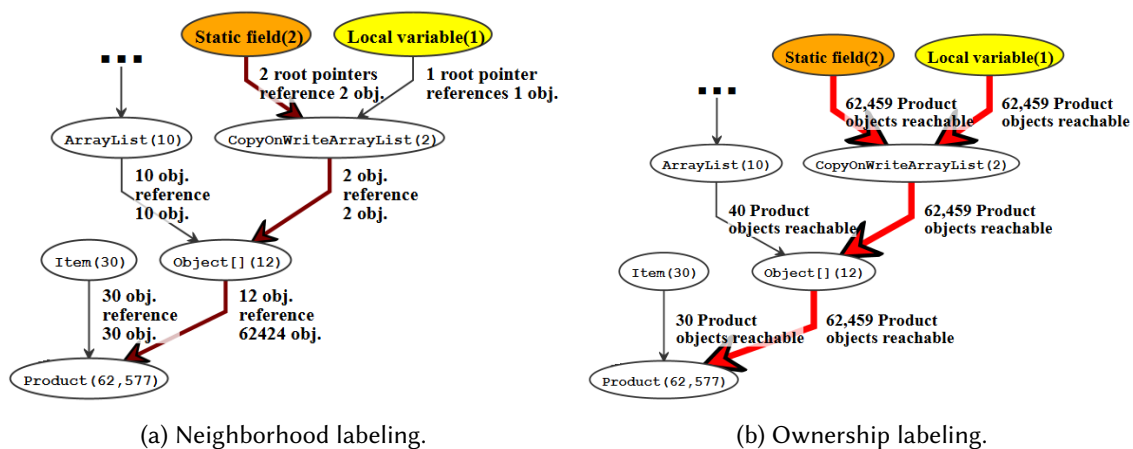


Fig. 4. Comparison of two edge labeling techniques.

9.5 Guidance

Many study participants expressed the risk of not using the tool to its full potential due to their missing background in memory analysis. They suggested that the tool should provide guidance features such that even novice users could use it without prior training. Since this was the most requested feature by the study participants, we have meanwhile extended AntTracks with an advisor feature realizing our recommendation to *provide guidance and explanations to support exploratory learning of analysis capabilities*. This guidance system performs four support operations on each analysis step: it (1) *detects* and (2) *highlights* the most important information on the screen, (3) *explains* why this information is important, and (4) *suggests* which next steps are appropriate based on these findings. This way, even without prior tool experience, AntTracks now guides users through the whole analysis process. For example, on the overview and the details views, AntTracks now automatically detects suspicious time windows, highlights them, explains why these time windows are considered suspicious and which next analysis steps seem appropriate [118], additionally realizing our recommendation to *provide support for time selection in large time series*.

Yet, while AntTracks now may guide users to suspicious allocation sites, they still have to fix the memory bug in their IDE without any further guidance. We thus further plan to integrate parts of this guidance also into IDEs, closing the gap of guided analyses in AntTracks and unguided source code inspection in the IDE, following the recommendation to *provide IDE integration to guide diagnosis of memory bugs*.

10 THREATS TO VALIDITY

A threat to construct validity is that our results might be biased due to the system we used during the study. However, we selected JPetStore because it easy to understand and at the same time allows to comprehensively assess the key views of AntTracks. Furthermore, JPetStore has also been used in many other studies [32, 53, 55, 56, 114] since it represents a clearly organized and realistic web application. Another threat to construct validity is the selection of appropriate tasks for our user study. To select representative memory analysis tasks, we discussed typical memory analysis activities by studying related work and state-of-the-art tools and selected the study tasks based on these activities. We further conducted a pilot study with a PhD student from our lab to further improve the expressiveness of the tasks.

To ensure external validity, we selected an interactive tool that is representative for the domain of memory monitoring and analysis. Although the implications we derived from the study depend on our experiences in using and working with AntTracks, the activities and capabilities are common in other memory analysis tools, as discussed in Section 3. Another threat is the use of students as subjects. They were selected based on their participation in a university course that teaches the basics of monitoring and performance analysis and includes a homework assignment on memory analysis. This allowed us to ensure that all subjects have a minimum of background knowledge on the context of the study. We have also shown that most of the subjects are software developers with several years of experience. Furthermore, it has been argued that the differences between students and professionals are only minor if they perform relatively small tasks of judgment [48]. While the participation of additional experienced memory analysts may have added further value, we realized that the findings for our subjects nicely converge and that there were many common insights. When qualitatively analyzing our results we noticed a certain degree of saturation and the findings showed that adding more novice participants would not have led to more insights.

Regarding internal validity, the analysis of the collected data still depends on our own interpretation. However, this work was performed by three researchers which had regular joint meetings to reconcile different interpretations, also checking their results with a fourth senior researcher.

With regard to conclusion validity, the threat consists in the fact that our results are based on qualitative data [104]. Given that an aim of the study was to investigate the behavior and opinions of tool users, qualitative research methods are well-suited. We further applied the derived recommendations to AntTracks to additionally check their validity.

11 CONCLUSIONS

This paper presented the detailed results of assessing the usefulness of memory analysis capabilities as implemented in the AntTracks Analyzer tool using the cognitive dimensions (CD) of notations framework and a user study involving 14 subjects. The CD framework serves as a discussion tool for designers and people who evaluate designs of interactive artifacts, including software tools. We used the results of the CD assessment and the user study to discuss both specific implications for AntTracks as well as general recommendations for memory analysis tool developers. Practitioners and researchers can use our work as one example of how to assess the usability and utility of interactive memory analysis tools.

Overall, we can conclude that the usability and utility of AntTracks were perceived as very good. All subjects liked the tool and most of them were able to complete the tasks within the planned time. However, our observations revealed some issues with regard to diagnosing and fixing problems on the source code level. Overall, the positive feedback is encouraging given the participants' limited background in memory analysis, their missing familiarity with the tool, and the complexity of the given analysis tasks. The comments and suggestions made by the study participants have been of great help to further improve our tool. For example, the guidance and support of novice users requested by many participants is a major focus for our ongoing work. First steps in this direction have already been taken since the study by introducing a guidance system that assists users during memory analysis. For example, this system automatically detects suspicious time windows, highlights them and explains why these time windows are considered suspicious [118]. This frees the users from hard mental operations, teaches them why certain patterns are of interest, and thus leads to a learning-by-doing effect [103].

In the future we will further improve our tool, especially with regard to novice user support, and conduct further experiments with other systems and users.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

- [1] Alain Abran, Adel Khelifi, Witold Suryn, and Ahmed Seffah. 2003. Usability Meanings and Interpretations in ISO Standards. *Software Quality Journal* 11, 4 (2003), 325–338. <https://doi.org/10.1023/A:1025869312943>
- [2] Edward Aftandilian, Sean Kelley, Connor Gramazio, Nathan P. Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010*. 53–62. <https://doi.org/10.1145/1879211.1879222>
- [3] Audacity. 2020. *Audacity: Free, open source, cross-platform audio software*. <https://www.audacityteam.org/>
- [4] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking sketches and diagrams to source code artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-22)*. 743–746. <https://doi.org/10.1145/2635868.2661672>
- [5] Matt Bellingham, Simon Holland, and Paul Mulholland. 2014. A cognitive dimensions analysis of interaction design for algorithmic composition software. In *Proceedings of the 25th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2014, Brighton, UK, June 25-27, 2014*. 18. <http://ppig.org/library/paper/cognitive-dimensions-analysis-interaction-design-algorithmic-composition-software>

- [6] Verena Bitto and Philipp Lengauer. 2016. Building Custom, Efficient, and Accurate Memory Monitoring Tools for Java Applications. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. 321–324. <https://doi.org/10.1145/2851553.2858664>
- [7] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015, Melbourne, FL, USA, September 8-11, 2015*. 76–89. <https://doi.org/10.1145/2807426.2807433>
- [8] Alan Blackwell and Thomas Green. 2003. CHAPTER 5 - Notational Systems—The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*. Morgan Kaufmann, San Francisco, 103 – 133. <https://doi.org/10.1016/B978-155860808-5/50005-8>
- [9] Alan F. Blackwell. 2005. Cognitive Dimensions of Notations. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), 21-24 September 2005, Dallas, TX, USA*. 3. <https://doi.org/10.1109/VLHCC.2005.26>
- [10] Alan F. Blackwell. 2008. Cognitive Dimensions of Notations: Understanding the Ergonomics of Diagram Use. In *Diagrammatic Representation and Inference, 5th International Conference, Diagrams 2008, Herrsching, Germany, September 19-21, 2008. Proceedings*. 5–8. https://doi.org/10.1007/978-3-540-87730-1_4
- [11] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and R. Michael Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings*. 325–341. https://doi.org/10.1007/3-540-44617-6_31
- [12] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 61–72. <https://doi.org/10.1145/1168857.1168866>
- [13] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. 1999. The Unified Modeling Language User Guide. *J. Database Manag.* 10, 4 (1999), 51–52.
- [14] Eric Bruneton, Eugene Kuleshov, Andrei Loskutov, and Rémi Forax. 2020. ASM. <https://asm.ow2.io/>
- [15] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).
- [16] Kung Chen and Ju-Bing Chen. 2007. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 2*. 23–28. <https://doi.org/10.1109/COMPSAC.2007.79>
- [17] Shigeru Chiba. 2020. *Javassist*. <https://www.javassist.org/>
- [18] Shigeru Chiba and Muga Nishizawa. 2003. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*. 364–376. https://doi.org/10.1007/978-3-540-39815-8_22
- [19] Adriana E. Chis. 2008. Automatic detection of memory anti-patterns. In *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*. 925–926. <https://doi.org/10.1145/1449814.1449911>
- [20] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O’Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of Memory Inefficiency. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. 383–407. https://doi.org/10.1007/978-3-642-22655-7_18
- [21] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. Performance-Hat: augmenting source code with runtime performance traces in the IDE. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings (ICSE 2018)*. 41–44. <https://doi.org/10.1145/3183440.3183481>
- [22] Karl Cox. 2000. Cognitive Dimensions of Use Cases: Feedback from a student questionnaire. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*. 8. <http://ppig.org/library/paper/cognitive-dimensions-use-cases-feedback-student-questionnaire>
- [23] Fred D. Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* 13, 3 (1989), 319–340. <http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html>
- [24] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the Execution of Java Programs. In *Software Visualization*. 151–162. https://doi.org/10.1007/3-540-45875-1_12
- [25] Wim De Pauw and Gary Sevitsky. 2000. Visualizing reference patterns for solving memory leaks in Java. *Concurrency - Practice and Experience* 12, 14 (2000), 1431–1454. [https://doi.org/10.1002/1096-9128\(200012\)12:14<1431::AID-CPE542>3.0.CO;2-2](https://doi.org/10.1002/1096-9128(200012)12:14<1431::AID-CPE542>3.0.CO;2-2)
- [26] D. Christopher Dryer. 1997. Wizards, Guides, and beyond: Rational and Empirical Methods for Selecting Optimal Intelligent User Interface Agents. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces, IUI 1997, Orlando, Florida, USA, January 6-9, 1997*. 265–268. <https://doi.org/10.1145/238218.238347>

- [27] Dynatrace. 2017. *Demo Applications: easyTravel*. <https://community.dynatrace.com/community/display/DL/Demo+Applications++easyTravel>
- [28] Dynatrace. 2020. *Dynatrace*. <https://www.dynatrace.com/>
- [29] Eclipse Foundation. 2020. *Eclipse Memory Analyzer (MAT)*. <https://www.eclipse.org/mat/>
- [30] ej technologies. 2020. *JProfiler*. <https://www.ej-technologies.com/products/jprofiler/overview.html>
- [31] Michael D. Ernst. 2003. Static and Dynamic Analysis: Synergy and Duality. In *Workshop on Dynamic Analysis (WODA '03)*. Portland, OR, USA, 24–27.
- [32] Florian Fittkau, Phil Stelzer, and Wilhelm Hasselbring. 2014. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *Proceedings of the ECSA 2014 Workshops & Tool Demos Track, European Conference on Software Architecture, 2014, Vienna, Austria*. 28:1–28:4. <https://doi.org/10.1145/2642803.2642831>
- [33] Eelke Folmer and Jan Bosch. 2003. Usability patterns in software architecture. In *Proc. of the 10th Int'l Conf. on Human-Computer Interaction (HCI '03)*. 93–97.
- [34] Tak-Chung Fu. 2011. A review on time series data mining. *Eng. Appl. Artif. Intell.* 24, 1 (2011), 164–181. <https://doi.org/10.1016/j.engappai.2010.09.007>
- [35] Mohamadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* 25, 1 (2020), 678–718. <https://doi.org/10.1007/s10664-019-09731-8>
- [36] David Gilbert. 2020. *JFreeChart*. <http://www.jfree.org/jfreechart/>
- [37] Google. 2020. *Android Studio*. <https://developer.android.com/studio>
- [38] Bernhard Göschlberger and Peter A. Bruck. 2017. Gamification in mobile and workplace integrated microlearning. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services, iiWAS 2017, Salzburg, Austria, December 4-6, 2017*. 545–552. <https://doi.org/10.1145/3151759.3151795>
- [39] Thomas Green. 2000. Instructions and Descriptions: some cognitive aspects of programming and similar activities. In *Proceedings of the working conference on Advanced visual interfaces, AVI 2000, Palermo, Italy, May 23-26, 2000*. 21–28. <https://doi.org/10.1145/345513.345233>
- [40] Thomas Green and Alan Blackwell. 1998. *Cognitive Dimensions of Information Artefacts: a tutorial*. <https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf>
- [41] T. R. G. Green. 1989. Cognitive Dimensions of Notations. In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*. Cambridge University Press, New York, NY, USA, 443–460. <http://dl.acm.org/citation.cfm?id=92968.93015>
- [42] Ted Hagos. 2019. Android Studio Profiler. In *Android Studio IDE Quick Reference*. Springer, 73–82.
- [43] Juho Hamari, Jonna Koivisto, and Harri Sarsa. 2014. Does Gamification Work? - A Literature Review of Empirical Studies on Gamification. In *47th Hawaii International Conference on System Sciences, HICSS 2014, Waikoloa, HI, USA, January 6-9, 2014*. 3025–3034. <https://doi.org/10.1109/HICSS.2014.377>
- [44] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*. 156–164. <https://doi.org/10.1145/1024393.1024412>
- [45] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. 2006. Generating object lifetime traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 476–516. <https://doi.org/10.1145/1133651.1133654>
- [46] Trent Hill, James Noble, and John Potter. 2002. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.* 13, 3 (2002), 319–339. <https://doi.org/10.1006/jvlc.2002.0238>
- [47] Andreas Holzinger. 2005. Usability engineering methods for software developers. *Commun. ACM* 48, 1 (2005), 71–74. <https://doi.org/10.1145/1039539.1039541>
- [48] Martin Höst, Björn Regnell, and Claes Wohlin. 2000. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering* 5, 3 (01 Nov 2000), 201–214.
- [49] Michal Hucko, Ladislav Gazo, Peter Simún, Matej Valky, Róbert Móra, Jakub Simko, and Mária Bielíková. 2019. YesElf: Personalized Onboarding for Web Applications. In *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization, UMAP 2019, Larnaca, Cyprus, June 09-12, 2019*. 39–44. <https://doi.org/10.1145/3314183.3324978>
- [50] Monique W. M. Jaspers, Thiemo Steen, Cor van den Bos, and Maud M. Geenen. 2004. The think aloud method: a guide to user interface design. *I. J. Medical Informatics* 73, 11-12 (2004), 781–795. <https://doi.org/10.1016/j.ijmedinf.2004.08.003>
- [51] JavaMelody. 2020. *JavaMelody : monitoring of JavaEE applications (GitHub)*. <https://github.com/javamelody/javamelody/wiki>
- [52] Kamil Jezek and Richard Lipka. 2017. Antipatterns causing memory bloat: A case study. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*.

- 306–315. <https://doi.org/10.1109/SANER.2017.7884631>
- [53] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated performance analysis of load tests. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. 125–134. <https://doi.org/10.1109/ICSM.2009.5306331>
- [54] Maria Jump and Kathryn S. McKinley. 2010. Detecting memory leaks in managed languages with Cork. *Softw., Pract. Exper.* 40, 1 (2010), 1–22. <https://doi.org/10.1002/spe.945>
- [55] Reiner Jung and Marc Adolf. 2018. The JPetStore Suite: A concise Experiment Setup for Research. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.
- [56] Reiner Jung, Marc Adolf, and Christoph Dornieden. 2017. Towards Extracting Realistic User Behavior Models. *Softwaretechnik-Trends* 37, 3 (2017). http://pi.informatik.uni-siegen.de/stt/37_3/.01_Fachgruppenberichte/SSP2017_proceedings/03_Towards_Extracting_Realistic_User_Behavior_Models.pdf
- [57] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. 327–353. https://doi.org/10.1007/3-540-45337-7_18
- [58] Kieker Project. 2013. *Kieker web site*. <http://kieker-monitoring.net/>
- [59] A. J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141. <https://doi.org/10.1007/s10664-013-9279-3>
- [60] Steinar Kristoffersen. 2008. Learnability and Robustness of User Interfaces - Towards a Formal Analysis of Usability Design Principles. In *Proceedings of the Third International Conference on Software and Data Technologies, Volume SE/MUSE/GSDCA (ICSOF08)*, José Cordeiro, Boris Shishkov, Alpesh Ranchordas, and Markus Helfert (Eds.). 261–268.
- [61] Lisa Maria Kritzing, Thomas Krismayer, Rick Rabiser, and Paul Grünbacher. 2019. A User Study on the Usefulness of Visualization Support for Requirements Monitoring. In *7th IEEE Working Conference on Software Visualization*. IEEE, Cleveland, Ohio, USA, 56–66. <https://doi.org/10.1109/VISSOFT.2019.00015>
- [62] Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development* (2007).
- [63] Maria Kutar, Carol Britton, and Jonathan Wilson. 2000. Cognitive Dimensions: An experience report. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*. 7. <http://ppig.org/library/paper/cognitive-dimensions-experience-report>
- [64] Maria Kutar, Chrystopher L. Nehaniv, Carol Britton, and Sara Jones. 2001. The Cognitive Dimensions of an Artifact vis-à-vis Individual Human Users: Studies with Notations for the Temporal Specification of Interactive Systems. In *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings*. 342–355. https://doi.org/10.1007/3-540-44617-6_32
- [65] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 4:1–4:11. <https://doi.org/10.1145/2972206.2972220>
- [66] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. 51–62. <https://doi.org/10.1145/2668930.2688037>
- [67] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. 249–260. <https://doi.org/10.1145/2851553.2851555>
- [68] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*. 3–14. <https://doi.org/10.1145/3030207.3030211>
- [69] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141. <https://doi.org/10.1145/357062.357071>
- [70] Beinan Li, John Ashley Burgoyne, and Ichiro Fujinaga. 2006. Extending Audacity for Audio Annotation. In *ISMIR 2006, 7th International Conference on Music Information Retrieval, Victoria, Canada, 8-12 October 2006, Proceedings*. 379–380.
- [71] João Paulo Magalhães and Luís Moura Silva. 2013. Adaptive monitoring of web-based applications: a performance study. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. 471–478. <https://doi.org/10.1145/2480362.2480454>
- [72] David Mapelsden, John Hosking, and John Grundy. 2002. Design Pattern Modelling and Instantiation Using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT '02)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 3–11. <http://dl.acm.org/>

- [org/citation.cfm?id=564092.564094](https://doi.org/10.1145/1120212.1120318)
- [73] Mark Marron, César Sánchez, Zhendong Su, and Manuel Fähndrich. 2013. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Software Eng.* 39, 6 (2013), 774–786. <https://doi.org/10.1109/TSE.2012.69>
 - [74] Karen L. McGraw and Bruce A. McGraw. 1997. Wizards, Coaches, Advisors, and More: A Performance Support Primer. In *Human Factors in Computing Systems, CHI '97: Looking to the Future, Extended Abstracts, Atlanta, Georgia, USA, March 22-27, 1997*. 152–153. <https://doi.org/10.1145/1120212.1120318>
 - [75] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. 74–98. https://doi.org/10.1007/11785477_5
 - [76] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making Sense of Large Heaps. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009, Proceedings*. 77–97. https://doi.org/10.1007/978-3-642-03013-0_5
 - [77] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2010. Four Trends Leading to Java Runtime Bloat. *IEEE Software* 27, 1 (2010), 56–63. <https://doi.org/10.1109/MS.2010.7>
 - [78] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. 351–377. https://doi.org/10.1007/978-3-540-45070-2_16
 - [79] Nick Mitchell and Gary Sevitsky. 2007. The causes of bloat, the limits of health. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 245–260. <https://doi.org/10.1145/1297027.1297046>
 - [80] MyBatis. 2016. *JPetStore*. <http://mybatis.org/jpetstore-6/>
 - [81] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.* 89, 2 (2003), 44–66. [https://doi.org/10.1016/S1571-0661\(04\)81042-9](https://doi.org/10.1016/S1571-0661(04)81042-9)
 - [82] Jakob Nielsen. 1993. *Usability engineering*. Academic Press.
 - [83] Mie Nørgaard and Kasper Hornbæk. 2006. What do usability evaluators do in practice?: an explorative study of think-aloud testing. In *Proceedings of the Conference on Designing Interactive Systems, University Park, PA, USA, June 26-28, 2006*. 209–218. <https://doi.org/10.1145/1142405.1142439>
 - [84] Oracle. 2014. *Java Flight Recorder*. <https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170>
 - [85] Oracle. 2018. *JConsole*. <https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html>
 - [86] Oracle. 2018. *jmap*. <https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>
 - [87] Oracle. 2020. *The HotSpot Group*. <http://openjdk.java.net/groups/hotspot/>
 - [88] Oracle. 2020. *HPROF: A Heap/CPU Profiling Tool*. <https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html>
 - [89] Oracle. 2020. *Java Mission Control*. <https://openjdk.java.net/projects/jmc/>
 - [90] Oracle. 2020. *JVM Tool Interface Version 1.2*. <https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>
 - [91] Oracle. 2020. *VisualVM: All-in-One Java Troubleshooting Tool*. <https://visualvm.github.io/>
 - [92] Kelly O’Hair. 2004. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network, Developer Technical Articles & Tips* 28 (2004).
 - [93] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2019)*.
 - [94] Rick Rabiser, Paul Grünbacher, and Martin Lehofer. 2012. A qualitative study on user guidance capabilities in product configuration tools. In *IEEE/ACM International Conference on Automated Software Engineering, ASE’12, Essen, Germany, September 3-7, 2012*. 110–119. <https://doi.org/10.1145/2351676.2351693>
 - [95] Rick Rabiser, Michael Vierhauser, and Paul Grünbacher. 2016. Assessing the usefulness of a requirements monitoring tool: a study involving industrial software engineers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 122–131. <https://doi.org/10.1145/2889160.2889234>
 - [96] Derek Rayside and Lucy Mendel. 2007. Object ownership profiling: a technique for finding and fixing memory leaks. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 194–203. <https://doi.org/10.1145/1321631.1321661>
 - [97] Steven P. Reiss. 2009. Visualizing the Java heap to detect memory problems. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2009, Edmonton, Alberta, Canada, September 25, 2009*. 73–80. <https://doi.org/10.1109/VISSOFT.2009.5336418>
 - [98] Jan Renz, Thomas Staubitz, Jaqueline Pollack, and Christoph Meinel. 2014. Improving the Onboarding User Experience in MOOCs. In *Proc. of the 6th Int’l Conf. on Education and New Learning Technologies (EDULEARN ’14)*.

- [99] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2011. Elephant Tracks: generating program traces with object death records. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*. ACM, 139–142. <https://doi.org/10.1145/2093157.2093178>
- [100] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant tracks: portable production of complete and precise gc traces. In *International Symposium on Memory Management, ISMM 2013, Seattle, WA, USA, June 20, 2013*. ACM, 109–118. <https://doi.org/10.1145/2491894.2466484>
- [101] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. <https://doi.org/10.1007/s10664-008-9102-8>
- [102] Anthony Savidis and Nikos Koutsopoulos. 2011. Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features. In *Advances in Visual Computing - 7th International Symposium, ISVC 2011, Las Vegas, NV, USA, September 26-28, 2011. Proceedings, Part I*. 259–268. https://doi.org/10.1007/978-3-642-24028-7_24
- [103] Roger C Schank, Tamara R Berman, and Kimberli A Macpherson. 1999. Learning by doing. *Instructional-design theories and models: A new paradigm of instructional theory* 2, 2 (1999), 161–181.
- [104] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572. <https://doi.org/10.1109/32.799955>
- [105] Ben Shneiderman. 1996. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, USA, September 3-6, 1996*. IEEE Computer Society, 336–343. <https://doi.org/10.1109/VL.1996.545307>
- [106] Connie U. Smith and Lloyd G. Williams. 2000. Software performance antipatterns. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*. 127–136. <https://doi.org/10.1145/350391.350420>
- [107] Ken Soong, Xin Fu, and Yang Zhou. 2018. Optimizing New User Experience in Online Services. In *5th IEEE International Conference on Data Science and Advanced Analytics, DSAA 2018, Turin, Italy, October 1-3, 2018*. 442–449. <https://doi.org/10.1109/DSAA.2018.00057>
- [108] Vladimir Sor, Plumb Ou, Tarvo Treier, and Satish Narayana Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. 544–547. <https://doi.org/10.1109/ICSM.2013.92>
- [109] Vladimir Sor and Satish Narayana Srirama. 2014. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software* 96 (2014), 139–151. <https://doi.org/10.1016/j.jss.2014.06.005>
- [110] Doug Tidwell and Jeanette Fuccella. 1997. TaskGuides: Instant Wizards on the Web. In *The 15th Annual International Conference of Computer Documentation: Crossroads in Communication, SIGDOC 1997, Salt Lake City, Utah, USA, October 19-22, 1997*. 263–272. <https://doi.org/10.1145/263367.263401>
- [111] Valgrind Developers. 2020. *Valgrind*. <http://valgrind.org/>
- [112] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. 2009. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Technical Report TR-0921. Department of Computer Science, Kiel University, Germany. 27 pages pages.
- [113] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*. 247–248. <https://doi.org/10.1145/2188286.2188326>
- [114] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. 213–222. <https://doi.org/10.1109/ICSM.2011.6080788>
- [115] Markus Weninger et al. 2020. *AntTracks - Memory Monitoring using Accurate and Efficient Object Tracing for Java Applications*. <http://mevss.jku.at/AntTracks>
- [116] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing object reference graphs and garbage collection roots to detect memory leaks in offline memory monitoring. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*. 14:1–14:13. <https://doi.org/10.1145/3237009.3237023>
- [117] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*. 273–284. <https://doi.org/10.1145/3297663.3310297>
- [118] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of suspicious time windows in memory monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*. 95–104. <https://doi.org/10.1145/3357390.3361025>
- [119] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.

- [120] Markus Weninger, Paul Grünbacher, Huihui Zhang, Tao Yue, and Shaukat Ali. 2018. Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment. In *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. 21–30. <https://doi.org/10.1109/APSEC.2018.00016>
- [121] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*. 357–360. <https://doi.org/10.1145/3030207.3030236>
- [122] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 07-11, 2019*. 29–32. <https://doi.org/10.1145/3302541.3313100>
- [123] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2019. Memory Leak Visualization using Evolving Software Cities. In *Proc. of the 10th Symposium on Software Performance (SSP '19)*.
- [124] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*. 115–126. <https://doi.org/10.1145/3184407.3184412>
- [125] Nicholas P. Wilde. 1996. Using Cognitive Dimensions in the Classroom as a Discussion Tool for Visual Language Design. In *Conference on Human Factors in Computing Systems: Common Ground, CHI '96, Vancouver, BC, Canada, April 13-18, 1996, Conference Companion*. ACM, 187–188. <https://doi.org/10.1145/257089.257252>
- [126] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing. *ACM Trans. Softw. Eng. Methodol.* 23, 3 (2014), 23:1–23:50. <https://doi.org/10.1145/2560047>
- [127] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. 421–426. <https://doi.org/10.1145/1882362.1882448>
- [128] Guoqing (Harry) Xu and Atanas Rountev. 2008. Precise memory leak detection for java software using container profiling. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. 151–160. <https://doi.org/10.1145/1368088.1368110>
- [129] Guoqing (Harry) Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 160–173. <https://doi.org/10.1145/1806596.1806616>
- [130] S. Zaman, B. Adams, and A. E. Hassan. 2012. A Large Scale Empirical Study on User-Centric Performance Analysis. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 410–419. <https://doi.org/10.1109/ICST.2012.121>
- [131] Thomas Zimmermann and Andreas Zeller. 2001. Visualizing Memory Graphs. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*. 191–204. https://doi.org/10.1007/3-540-45875-1_15

Received October 2019; revised November 2019; accepted December 2019

6.3 Guided Exploration

This section includes the paper that presents our user guidance pipeline called *guided exploration* and how it supports users during memory leak and memory churn analysis.

Paper:

Markus Weninger, Elias Gander, Hanspeter Mössenböck:

Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools. In *Proceedings of the ACM on Human Computer Interaction*, Vol. 5 (EICS), June 2021. (in press)

Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools

MARKUS WENINGER, Institute for System Software, Johannes Kepler University Linz, Austria

ELIAS GANDER, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

HANSPETER MÖSSENBOECK, Institute for System Software, Johannes Kepler University Linz, Austria

Many monitoring tools that help developers in analyzing the run-time behavior of their applications share a common shortcoming: they require their users to have a fair amount of experience in monitoring applications to understand the used terminology and the available analysis features. Consequently, novice users who lack this knowledge often struggle to use these tools efficiently.

In this paper, we introduce the *guided exploration* (GE) method that aims to make interactive monitoring tools easier to use and learn. In general, tools that implement GE should provide four support operations on each analysis step: they should automatically (1) *detect* and (2) *highlight* the most important information on the screen, (3) *explain* why it is important, and (4) *suggest* which next steps are appropriate. This way, tools *guide* users through their analysis processes, helping them to *explore* the root cause of a problem. At the same time, users learn the capabilities of the tool and how to use them efficiently.

We show how GE can be implemented in new monitoring tools as well as how it can be integrated into existing ones. To demonstrate GE's feasibility and usefulness, we present how we extended the memory monitoring tool AntTracks to provided guided exploration support during memory leak analysis and memory churn analysis. We use these guidances in two user scenarios to inspect and improve the memory behavior of the monitored applications.

We hope that our contribution will help usability researchers and developers in making monitoring tools more novice-friendly by improving their usability and learnability.

CCS Concepts: • **General and reference** → *Design*; • **Software and its engineering** → Software system structures; *Dynamic analysis*; *Software performance*; **Software maintenance tools**; *Software design techniques*; *Software defect analysis*; Maintaining software; • **Human-centered computing** → Graphical user interfaces; **User centered design**; *User interface design*; *User interface programming*.

Additional Key Words and Phrases: Monitoring Tools, Guided Exploration, Advisor, Onboarding, Intelligent Assistant, Context-Sensitive Help, Program Comprehension, Memory Comprehension, Memory Monitoring, Memory Leak Analysis, Memory Churn Analysis

ACM Reference Format:

Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2021. Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools. *Proc. ACM Hum.-Comput. Interact.* 5, EICS, Article 209 (June 2021), 34 pages. <https://doi.org/10.1145/3461731>

Authors' addresses: Markus Weninger, markus.weninger@jku.at, Institute for System Software, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Elias Gander, elias.gander@jku.at, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Hanspeter Mössenböck, hanspeter.moessenboeck@jku.at, Institute for System Software, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria.

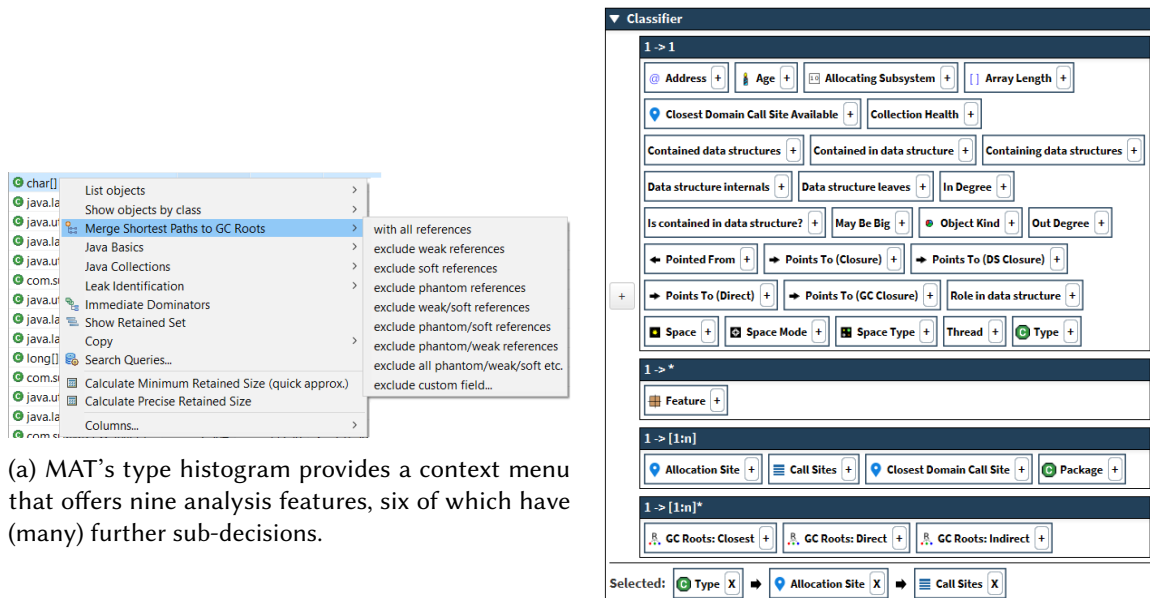
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the ACM on Human-Computer Interaction*, <https://doi.org/10.1145/3461731>.

1 INTRODUCTION

The complexity of modern software makes monitoring tools essential, as their analysis features support users in inspecting, understanding, and fixing run-time problems. Unfortunately, most monitoring tools are designed for experts with extensive knowledge in their respective domain. Consequently, novices who lack this expertise are often unable to use these tools to their full potential [57, 79]. For example, Weninger et al. [91] observed during a user study on memory monitoring tools that especially novice users who were unfamiliar with typical memory monitoring activities and tool features struggled to extract the insights needed to fix a given problem. They were often overwhelmed by the complexity and number of available features and said that they wished to have more guidance throughout the analysis process. Based on these observations, the authors recommend that memory monitoring tool developers should *provide guidance and explanations to support exploratory learning of analysis capabilities* [91].

Having too many and too complex features is a common problem across interactive expert tools, which makes them hard to use for novice users, as illustrated in Figure 1. For example, in the memory analysis tool MAT [80], to inspect a suspicious heap object group in more detail, users are confronted with a long list of analysis features (Figure 1a). In AntTracks [85, 99], users can define how to group heap objects for inspection based on a number of different properties and criteria (Figure 1b). Without guidance or hints, novice users may feel overwhelmed and may not be able to decide which actions to take in a certain situation.



(a) MAT's type histogram provides a context menu that offers nine analysis features, six of which have (many) further sub-decisions.

(b) AntTracks's classification view offers around 35 grouping criteria for heap objects that can be freely combined by the user.

Fig. 1. Examples of complex decisions in memory monitoring tools.

In this paper, we present *guided exploration* (GE), a method that aims to increase the learnability and usability of monitoring tools, and GE's application in memory monitoring. In general, to implement GE, tool developers should first identify their tool's typical analysis processes. For example, in a memory monitoring tool, this may be the typical steps performed during memory leak analysis. GE aims to support users in performing and understanding these steps, especially

supporting those users without extensive knowledge in the tool's domain. To do so, tools that follow the method of GE should provide the following four support operations on every analysis step: they should automatically (1) *detect* and (2) *highlight* the most important information on the screen, (3) *explain* background knowledge and why the highlighted information is important, and (4) *suggest* which next steps are appropriate based on the findings.

To demonstrate how GE can be introduced in existing tools, we present how we extended the interactive memory monitoring tool AntTracks to support GE on its two main analyses: memory leak analysis and memory churn analysis.

Our main contributions in this work are:

- (1) an overview of our general *guided exploration* method that can be implemented in new monitoring tools as well as integrated into existing ones, see [Section 3](#).
- (2) guided exploration for memory leak analysis, integrated into AntTracks, see [Section 4](#).
- (3) guided exploration for memory churn analysis, integrated into AntTracks, see [Section 5](#).
- (4) a discussion of preliminary user feedback regarding AntTracks's GE ([Section 6](#)), an outlook on the possible application of GE in a domain other than memory monitoring ([Section 7](#)), and a discussion of GE's current limitations and possible future improvements ([Section 8](#)).

2 BACKGROUND AND RELATED WORK

In this section, we first discuss background and related work in the field of Human-Computer Interaction, more specifically related work on the usability and ease of use of analysis and monitoring tools. We then show different kinds of user guidance and how our approach fits into these classifications. Since this work focuses on novice user guidance in memory monitoring tools, we also introduce general memory monitoring concepts and typical memory problems that developers have to face. As we have implemented guided exploration in the memory monitoring tool AntTracks, AntTracks and its core features are also explained.

2.1 Usability, Ease of Use and Learnability

Ample studies have been performed on how to improve the user experience in software tools. For example, Johnson et al. [33] performed a study on the (under-)use of static analysis tools. Nineteen of their 20 study participants reported that they *felt that many static analysis tools do not present [...] enough information for them to assess what the problem is, why it is a problem and what they should be doing differently*, i.e., they missed explanations on how to interpret the presented data. Christakis and Bird [10] report that many of their findings match those of Johnson et al. In a study conducted by Riemenschneider and Hardgrave [70], *ease of use* (including *learnability*) was shown to be the major determinant for *tool usage*, i.e., ease of use is paramount for tools to attract and hold users. Holding users is important, since a continuous use of monitoring tools, especially application performance management (APM) tools, can have a positive impact on the quality of software [77]. Despite this, Tarek et al. [2] conclude their work on the effectiveness of APM tools as *[...] the reporting capability of APM tools must be improved to reduce the effort that is required to analyze detected performance regressions*. The logical consequence that follows from these results is that developers have to improve their tools' usability to reach a broad range of users. While some approaches try to achieve this using user-specific data aggregation [64, 82], our GE approach focuses on increased learnability by *guiding* the user through analysis processes.

2.2 User Guidance

The idea of user guidance is not new. Folmer and Bosch [18] classify two general *guidance patterns* that are typically used to increase tool usability [1, 38, 54]: (1) *wizards* and (2) *context-sensitive help*.

Most *wizards* are implemented as a rigid, linear series of dialog views [12]. These views ask a number of questions and then use this information to automate certain tasks [81]. Modern approaches involve the generation of user-specific wizards [102].

Various approaches exist for *context-sensitive help* [78], such as *coaches* [50], *guides* [12] or *advisors* [50]. *Coaches* are often implemented as context-sensitive hints or tips and typically provide the user “how to” information to overcome minor hurdles. *Guides* can be thought of as “intelligent coaches”, as they only display hints or tips whenever and wherever it is most likely useful, reacting to the user’s behavior. Coaches, guides, and more recent approaches such as micro-learning and gamification [22, 23, 28] are often used during *onboarding* [67], i.e., while introducing a person to a new tool to improve the person’s success using it [76]. Our approach differs from coaches and guides as GE does not only explain possible next steps or display certain hints, but it provides full guidance throughout a given task, including automatic decision making based on the underlying data. Thus, it better fits the description of an *advisor* system. Advisors are context-sensitive help systems that provide hints, tips, reasoning support, and explanations of complicated concepts. They help novice users to make decisions, to understand why certain steps should be performed, and to determine why certain decisions were suggested.

Rabiser et al. [62] provide a framework to compare monitoring approaches based on 21 different characteristics, including typical characteristics relating to guidance such as *target group*, *needed skills*, *input guidance*, and *output guidance*. They then compared 32 existing monitoring approaches and tools based on this framework. Even though they report that some monitoring tools partly provide certain unstructured guidance, they conclude that *many approaches do not provide much end-user tool support [...] and generally only very few provide fully-fledged tools with visualizations and guidance for users. The target user group seems to be mainly (experienced) engineers* [62].

To the best of our knowledge we are the first to describes a general guidance method in the context of interactive monitoring tools, especially in the domain of memory monitoring.

2.3 Memory Analysis

To reduce the risk for memory-related problems, modern programming languages such as Java use *garbage collection* (GC) to automatically reclaim unused memory. During a garbage collection, objects that are no longer (indirectly) reachable from GC roots (i.e., static fields and local variables) are automatically reclaimed, freeing up their reserved memory. This relieves programmers from the error-prone task of manual memory management. Nevertheless, garbage collection comes with its own set of possible memory problems that can slow down applications if developers handle object allocations and object storage carelessly. In the worst case, problems such as memory leaks can even crash the application.

Memory leaks occur when objects that are no longer needed remain reachable from GC roots due to programming errors [48]. For example, a developer may forget to remove objects from long-living data structures once they are not needed anymore. These objects cannot be reclaimed by the GC and will therefore accumulate over time [86, 88].

Another common memory anomaly that is often overlooked by novice users is *high memory churn*. High memory churn, also called *excessive dynamic allocations* [60, 75] or *high allocation density* [13], occurs when objects are (unnecessarily) allocated in high frequencies, just to be reclaimed shortly after their creation. For example, high memory churn is often the result of heavily-executed loops that contain allocations of short-living objects. This leads to increased work for allocating these objects on the heap and to an increased number of garbage collections to collect them, both of which negatively impact an application’s performance.

2.4 Introduction to AntTracks

This section presents the basics of AntTracks, a trace-based memory monitoring tool consisting of the AntTracks VM [39–41] (a modified Java Hotspot VM) and the AntTracks Analyzer [5, 86–90, 93, 94, 99]. We use AntTracks as an example throughout the paper to showcase how existing monitoring tools can be extended and refactored to support GE. We chose this tool since its source code is publicly available [85] and the authors already had prior experience with its code base.

2.4.1 Trace Recording by the AntTracks VM. The AntTracks VM records events such as object allocations and object movements during garbage collection by writing them into trace files [39, 40], introducing a run-time overhead of about 5%. To reduce the trace size, the VM does not record any redundant data and applies compression [41].

2.4.2 Reconstruction in the AntTracks Analyzer. The AntTracks Analyzer processes the events stored in a trace file, reconstructing the heap state at every garbage collection point [5]. A heap state is a set of heap objects that were live in the monitored application at a certain point in time. Properties such as the the address, the type, the allocation site, and the allocating thread can be reconstructed for each heap object, as well as GC root information and information about the references between the heap objects.

The tool’s core mechanism is object classification in combination with multi-level grouping [93, 99]. A classifier groups heap objects according to a certain criterion such as type, allocation site, or allocating thread. Grouping the heap objects according to the classification results of multiple classifiers results in a hierarchical *memory tree*. A common classifier combination is to group all heap objects by their types and then by their allocation sites, as exemplarily shown in Figure 2. Yellow rectangles represent tree nodes and blue circles represent the objects that were classified into the respective tree branch. For example, the objects 0 to 3 are of type `Object[]`, of which the objects 0, 1 and 3 have been allocated in the method `Stack: init()` and object 2 has been allocated in the method `MyService: foo()`.

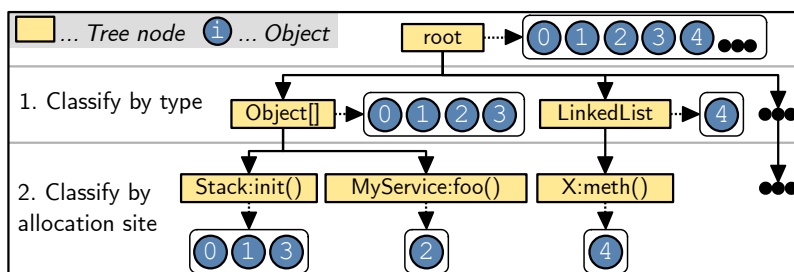


Fig. 2. A *memory tree* that first groups all objects by their types and then by their allocation sites.

Memory traces are used as a data basis for a variety of analyses within AntTracks. Two of these analysis, as well as their new guided exploration features, will be explained in more detail in Section 4 (memory leak analysis) and Section 5 (memory churn analysis).

3 GUIDED EXPLORATION

Without training, especially novice users may struggle to understand analysis features, terminology, metrics or visualizations in state-of-the-art interactive monitoring tools due to their steep initial learning curve. Being a novice monitoring tool user does not imply general inexperience. For example, even experienced software developers may have never used a memory monitoring tool before they encounter their first application crash due to a memory leak, which makes them a novice

in the domain of memory analysis. By incorporating learning-by-doing [71], our *guided exploration* method intends to simplify the onboarding process for this kind of users. As the method’s name suggests, tools implementing guided exploration should *guide* novice users through their analysis processes, helping them to *explore* the collected data until the root cause of a problem is found.

Even though the focus of this work is to present how GE can be applied in memory monitoring tools, this section presents the general idea of GE. As we discuss in Section 7, we think that GE may be a suitable guidance pattern for monitoring tools of other domains too, and thus we plan to further explore possible applications of GE in other domains in future work.

Section 3.1 discusses which steps are necessary before GE can be integrated into a tool. Section 3.2 explains GE’s four user support operations (depicted in Figure 4) in detail: *Detection*, *Highlighting*, *Explanation*, and *Suggestion*. These four user support operations can be gradually introduced in existing monitoring tools view by view, step by step.

3.1 Mapping of Analysis Process Steps to Views

Before introducing guided exploration in a monitoring tool, the tool developers have to define its typical analysis processes (such as memory leak analysis in a memory monitoring tool) and the steps performed within these processes. To do so, we suggest to create a (simple) process or task model. Various task model notations exist, for example ConcurTaskTrees [58], Task Flow [37], useML [51], or visualizations similar to UML statechart diagrams [46]. They all strive to capture the most important elements describing how a task (i.e., an activity that should be performed in order to reach a certain goal) is carried out by a particular user in a given context or in a given scenario [21, 45]. In the case of guided exploration, these task models should be designed from the perspective of novice users. This means that, even though analysis tasks (e.g., memory leak analysis) can often be performed in different ways across a number of multiple steps, the model should contain the *typical* flow of steps that should be performed to achieve the task’s goal. Tool developers and domain experts should be able to derive such a model, describing the “default” steps novice users should learn to perform, i.e., those steps that should be supported with GE in the future. Each step can then be mapped to one of the tool’s views to determine those views that have to be modified in order to support GE.

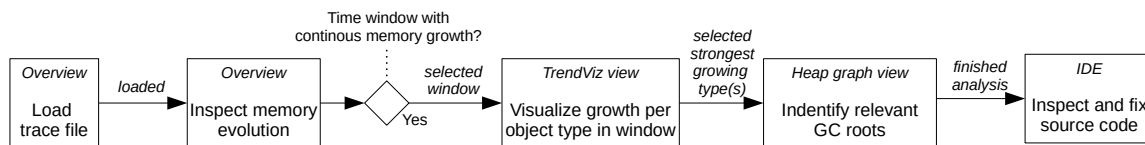


Fig. 3. Simplified task model of the typical steps performed during memory leak analysis, mapped to their corresponding AntTracks views.

For example, in Section 4 we show how GE has been implemented in AntTracks to guide users on the search for memory leaks. A typical memory leak analysis process, as shown in Figure 3, is (1) to search for a time window with continuous memory growth, (2) to find those objects that accumulate over time within this time window, and (3) to identify the GC roots that keep these accumulating objects alive. In AntTracks, each of these analysis steps is performed on a separate view, for each of which the four guided exploration support operations depicted in Figure 4 and explained in Section 3.2 have been implemented.

3.2 Guided Exploration Support Operations

This section discusses the four GE support operations a tool should perform on each analysis step: (1) First, the tool should automatically *detect* potential problems, i.e., suspicious patterns. (2) To help users in understanding from where the automatically gained insight was derived, the respective user interface (UI) region should be visually *highlighted*. (3) Since the user may require background knowledge to comprehend certain terminology and the highlighted information, *explanations* should discuss why the highlighted information is interesting. (4) Finally, based on the problem and the detected information, subsequent analysis steps should be *suggested*.

Since tools can greatly differ in their look-and-feel, we did not come up with a general rule on how to visualize notifications that a suspicious pattern has been found. It is thus up to the tool developer to appropriately inform the user about new guidance information. For example, in AntTracks, if guidance information is available a *guidance button* in the form of a light bulb is shown next to the respective UI element. Clicking such a light bulb then *highlights* the respective UI element and provides *explanations* and *suggestions*. This way of visualization was developed based on preliminary user feedback, as we will discuss in Section 6. Also, tool developers should keep guidance support optional, as experienced users may prefer to perform inspections without guidance elements visible.

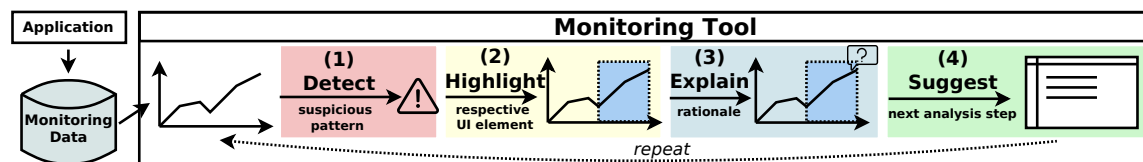


Fig. 4. The four guidance operations of GE: (1) Detection, (2) Highlighting, (3) Explanation, and (4) Suggestion.

Detection describes the task of automatically detecting potential problems, i.e., suspicious metrics or patterns.

Every view in an interactive monitoring tool is developed with the intention of supporting the user in achieving a certain goal. To this end, different kinds of visualizations are used to present data to the user. In non-guiding tools, it is up to the user to interpret these visualizations and to derive insights and findings from them. For example, a tool may present time-series charts for users to detect abnormal program behavior patterns. Others present tables, expecting the user to search for suspicious entries, e.g., metrics exceeding certain thresholds.

Most of these tasks require domain expertise that novice users generally do not have. For example, novice memory monitoring tool users may not search for data structures with a large *retained size*, i.e., the number of objects that are kept alive by a data structure [87], if this definition is unknown to the user. Thus, the first support operation of GE is to automatically *detect* suspicious patterns that may hint at problems in the monitored application. We think that intelligent tools should be able to perform this task, at least to a certain degree. After all, non-guiding tools expect their users to be able to detect suspicious patterns on their own, based on the displayed information. Since both the tool and the user have access to the same information, the tool should be capable of performing the same detection task, even if only detecting more obvious patterns using heuristics.

Even though the problem patterns that need to be detected may differ from domain to domain, we found certain similarities across different monitoring tools regarding the data they operate on and how this data is presented to users. Many monitoring tools inspect the *evolution* of a system, i.e., the evolution of certain metrics over time. These metric changes are often visualized

using charts, primarily time-series charts. Users are then expected to detect suspicious patterns within this evolution. Automated time series analysis [19] (using features such as regression analysis [53], seasonal and trend analysis [84], prediction [73], forecasting [4], or clustering and anomaly detection [43]) is a major research field in the domain of knowledge discovery and data mining. For example, time series analysis is used in memory monitoring to automatically detect suspicious time windows during which a monitored application behaves abnormally with regard to memory utilization [89].

Another typical way of depicting information is through the use of (hierarchical) tables, where features such as filtering or sorting should help the user to detect entries that (do not) meet certain criteria or those that exceed given thresholds. Such tasks may also be supported by automatic detection algorithms. For example, in the domain of memory monitoring, some memory inefficiencies and anti-patterns can automatically be detected based on memory metrics that exceed certain limits [8, 9]. *Intelligent user interfaces* [24, 30, 34, 49] often also apply artificial intelligence and machine learning for pattern detection, for example, by performing clustering or outlier detection. Also, research in the field of recommender systems [16, 65] may provide interesting ideas and algorithms that could be incorporated into the automated detection of patterns in monitoring tools.

Highlighting the relevant region on the user interface helps users to understand *where* the automatically gained insight can be found if the view was inspected manually.

The goal of GE is *not* to remove visualizations from monitoring tools, but to help users to understand and interpret these visualizations better. Thus, once a potential problem is detected, the UI region / element relevant for its detection should be *highlighted*. Different types and arrangements of UI elements may use different kinds of highlighting. The kind of highlighting should be chosen based on known UI design principles such as the principles of highlighting [44] or color coding [100]. Further, the developers should make sure that the style of highlighting is *consistent* throughout all views that support guided exploration [6]. For example, Figure 5 shows how AntTracks highlights rows in its tree tables by displaying them with a different background color.





Name	Objects	Retained size ▾
▾ Overall	8,022,993	404 MB
▾ Filtered	429,915	172.7 MB
▾  HashMap	68,214	133.4 MB
 SetMultimap::<init>()	6	91.8 MB
▶  AllocatedTypes::<init>()	3	35.2 MB
▶  SymbolsParser::parseAllo...	64,720	11.8 MB

Fig. 5. AntTracks now automatically *detects* and *highlights* suspicious parts, for example data structures that keep many other objects alive (i.e., those data structures that have a high retained size).

Explanations should help users in understanding why the highlighted information is important. They should clarify used terminology and concepts that are needed to understand the problem.

Let's continue with the example from Figure 5. First, AntTracks automatically *detects* objects with a high retained size [87], i.e., objects that keep a large number of other objects alive, and then

highlights these objects on the view (in this case six HashMap objects allocated at the same allocation site). Without knowing what a high retained size means or how to interpret it, the user will not be able to make sense of the highlighted information. Thus, the user can choose to display an *explanation* that should clarify needed *background knowledge*, *terminology*, as well as the *rationale* why the given pattern is considered suspicious.

Figure 6 exemplarily shows how AntTracks handles this in its guidance pop-ups. First, the explanation describes what retained size means (background knowledge + terminology), followed by an explanation of the currently highlighted area, i.e., “Over 22% of this heap is kept alive by 6 data structures of type HashMap that have been allocated in the constructor of class SetMultimap” (rationale).

Suggestions on which steps could or should be taken next to make it easier for the user to understand *what* operations are possible and *why* they might be useful.

Interactive monitoring tools often provide a vast amount of analysis features that can be applied in different situations. Intended for expert users, this flexibility may intimidate and overwhelm novice users. Despite the multitude of available features, as discussed in Section 3.1, most analysis processes have a default flow of tasks. *Suggestions* should guide the user through this process. We also recommend to not only display these suggestions as plain text, but to provide shortcuts to automatically perform the suggested actions. For example, Figure 6 shows how AntTracks presents suggested operations as buttons that automatically perform the next step.

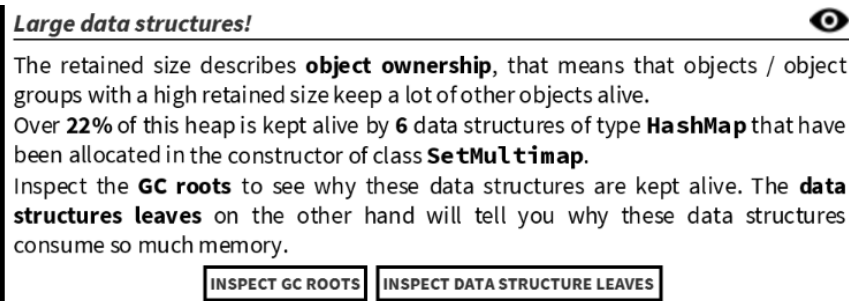


Fig. 6. This *explanation* and *suggestion* pop-up is shown upon user request in AntTracks when data structures with large ownership are detected. It explains terminology, e.g., retained size, and explains which insights might be gained following the provided suggestions.

4 GUIDED EXPLORATION OF MEMORY LEAKS

In this section, we present how we integrated *guided exploration* into AntTracks to facilitate the analysis of memory leaks. In Section 4.1, we describe a typical memory leak analysis process and how this process is mapped to AntTracks’s views. In Section 4.2, we describe how AntTracks’s views have been extended to support the four guided exploration operations *Detection*, *Highlighting*, *Explanation*, and *Suggestion*. To showcase how these new guidances support users in comprehending and investigating memory leaks, we use AntTracks’s new guided exploration and its suggestions to investigate and fix an application that contains a memory leak.

4.1 Mapping of Memory Leak Analysis Process Steps to Views

There are various ways to detect and analyze memory leaks. For example, AntTracks can detect memory leaks by searching for growing data structures and inspecting those with the strongest growth in more detail [86, 88]. Thus, we extended AntTracks’s data structure growth analysis by implementing guided exploration for it.

Yet, not every memory monitoring tool can access data structure information in the monitored application. Thus, we will focus on a more general memory leak analysis process that is not specific to AntTracks, which is visualized as a simplified task model in Figure 7. It consists of the following three steps:

- (1) Detect a time window with continuous memory growth, i.e., a continuous time frame in which object accumulate over time.
- (2) Find out which kinds of objects accumulate over time in this window.
- (3) Find those GC roots that keep these strongly accumulating objects alive.

In AntTracks, each step is performed on a different view:

- (1) The *Overview view* plots the application’s memory evolution and GC activity in time-series charts. A growing number of heap objects over time may hint at a possible memory leak.
- (2) The *AntTracks TrendViz view* [94] shows how the heap evolves over time, i.e., which objects (grouped by, e.g., their types) accumulate the most.
- (3) The *Heap graph view* interactively visualizes a heap state in a graph-based visualization. It can be used to inspect *keep-alive* relations to drill-down to the root cause of a possible memory leak.

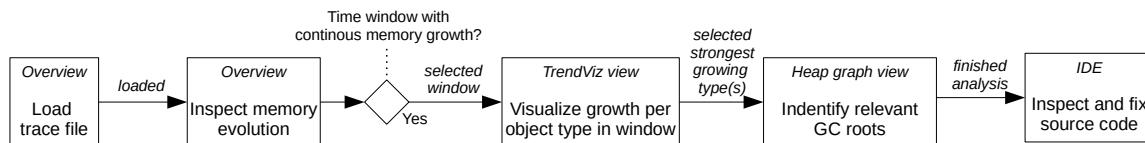


Fig. 7. Simplified task model of the typical steps performed during memory leak analysis, mapped to their corresponding AntTracks views.

The first two views have already existed in AntTracks and have been extended to provided guided exploration as part of this work. The heap graph view has been newly developed from scratch, including its GE support.

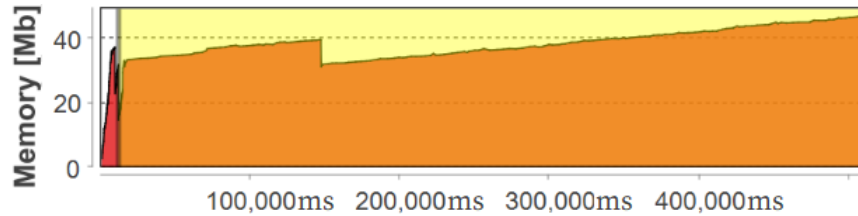
The analysis steps shown in Figure 7 are not restricted to AntTracks but can also be performed in a similar way in other memory monitoring tools such as VisualVM [56] or MAT [80]. Thus, the GE support operations that have been integrated into AntTracks could be integrated into these tools in a similar fashion as well.

4.2 Guided Exploration Support Operations for Memory Leak Analysis

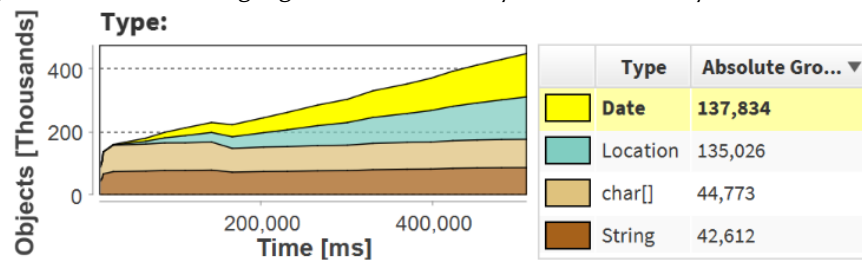
AntTracks now provides GE on the three views identified in Section 4.1. In this section, for each view we explain its general functionality and its new GE support operations *Detection*, *Highlighting*, *Explanation* and *Suggestion*.

To showcase how GE in AntTracks now supports users in comprehending and investigating memory leaks, we present how the newly introduced guidance features have been used to inspect *Dynatrace easyTravel* [14]. Dynatrace focuses on application performance monitoring (APM) and distributes easyTravel as their state-of-the-art memory leak demo application. It is a multi-tier application for a travel agency, using a Java backend and an automatic load generator simulating

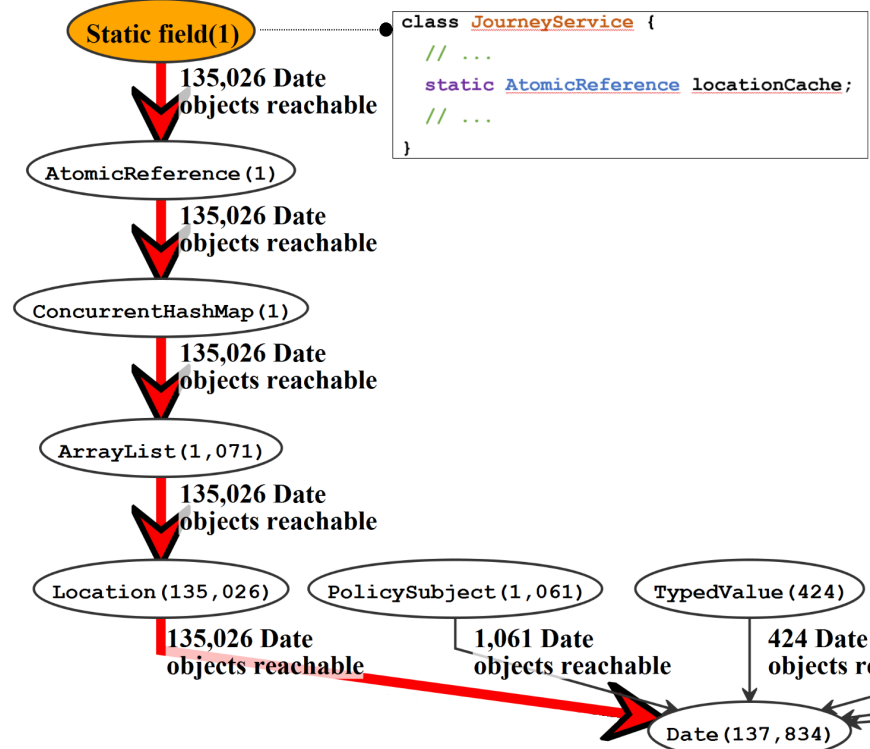
accesses to the service. All automatically detected and highlighted problem patterns are shown in Figure 8 and will be explained in detail in the following.



(a) The overview view highlights an automatically detected memory leak time window.



(b) AntTracks TrendViz shows how objects of various types accumulate over time.



(c) The graph view highlights the path from a selected group of objects (i.e., Date) to its most important garbage collection root (i.e., the static field locationCache), i.e., the path on which most Date objects are potentially kept alive.

Fig. 8. Memory leak analysis in AntTracks.

4.2.1 Overview View.

The overview (Figure 9) view gives the user a general impression on the application's memory behavior. For example, a time-series chart plots the monitored application's memory footprint over time. Users can select a single point in time to inspect the heap state at that point, or they can select a time window, i.e., two points in time, to inspect the heap evolution over this window.

We observed at different occasions, e.g., during studies or when AntTracks was used during hands-on tool presentations, that especially novice users are in need of guidance and support. Some users lacked the background knowledge to recognize abnormal behavior as such, or they struggled to select a suitable point in time or a suitable time window for certain analysis features. GE on AntTracks's overview view should help the users by automating these steps.

Detection. Weninger et al. [89] showed how to automatically detect suspicious time windows in memory monitoring. We apply their heuristic-based algorithm that mimics human behavior to search for a memory leak window, i.e., a window with a continuous growth (except for minor drops) of reachable memory.

Highlighting. If a suspicious time window that may be the result of a possible memory leak is found, it is highlighted with a yellow rectangular overlay, as shown in Figure 8a.

Explanation. AntTracks explains to the user why the detected time window may be connected to a memory leak: *AntTracks has detected a time window over which the reachable memory is continuously growing. This is an indicator for a possible memory leak. If a memory leak exists, typically objects of a few common types accumulate over time.*

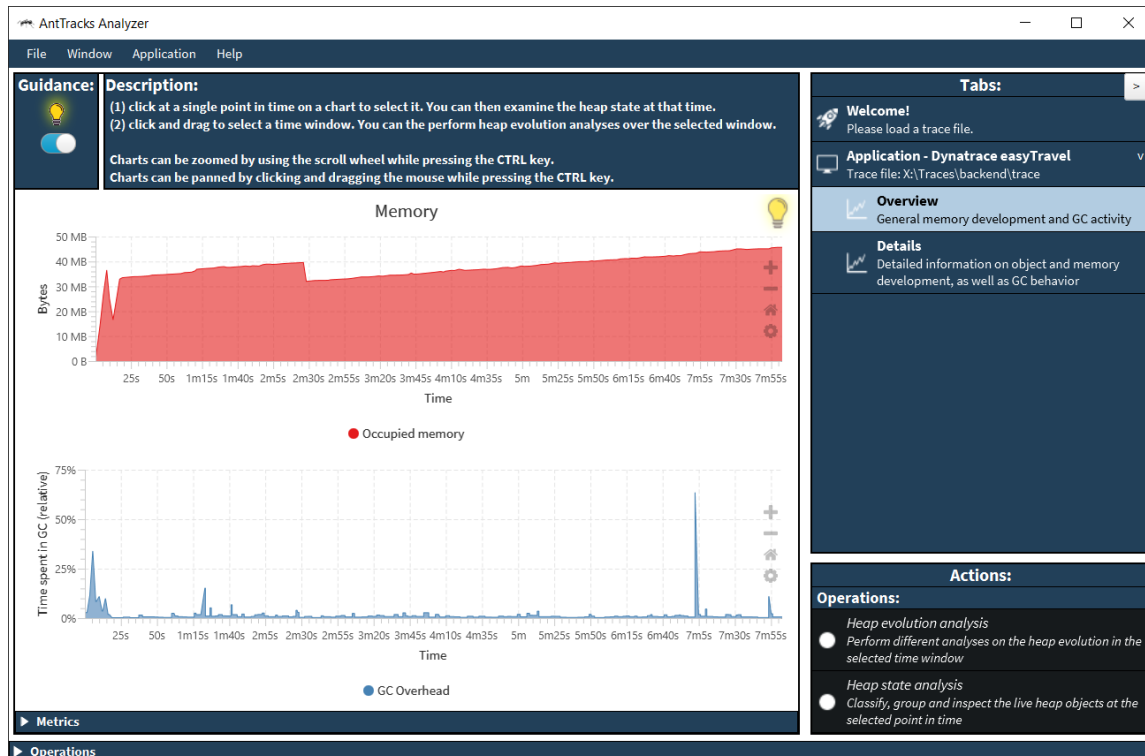


Fig. 9. The Overview provides initial information to assess the overall memory and garbage collection behavior of the monitored application.

Suggestion. We suggest the user to apply the *AntTracks TrendViz* [94] feature to explore how the heap's contents changed over time in order to detect if certain types of objects accumulate over time.

Figure 8a shows the memory evolution of easyTravel over time, including an automatically detected and highlighted memory leak time window. AntTracks explains that the window exhibits strong memory growth (about 400%) up until the end of the application, an indication for a memory leak. It is worth mentioning that the initial memory spike during application startup is not part of this window. Objects allocated during this spike are freed shortly after and thus have no relevance for the memory leak, a fact that is obvious for experts (and the time window algorithm) but novice users may not be aware of. Following the suggestion, we applied the *AntTracks TrendViz* feature on the time window.

4.2.2 AntTracks TrendViz View.

The AntTracks TrendViz view [94] (Figure 10) classifies the live heap objects at every garbage collection based on a list of selected classifiers, as explained in Section 2.4. The evolution of the

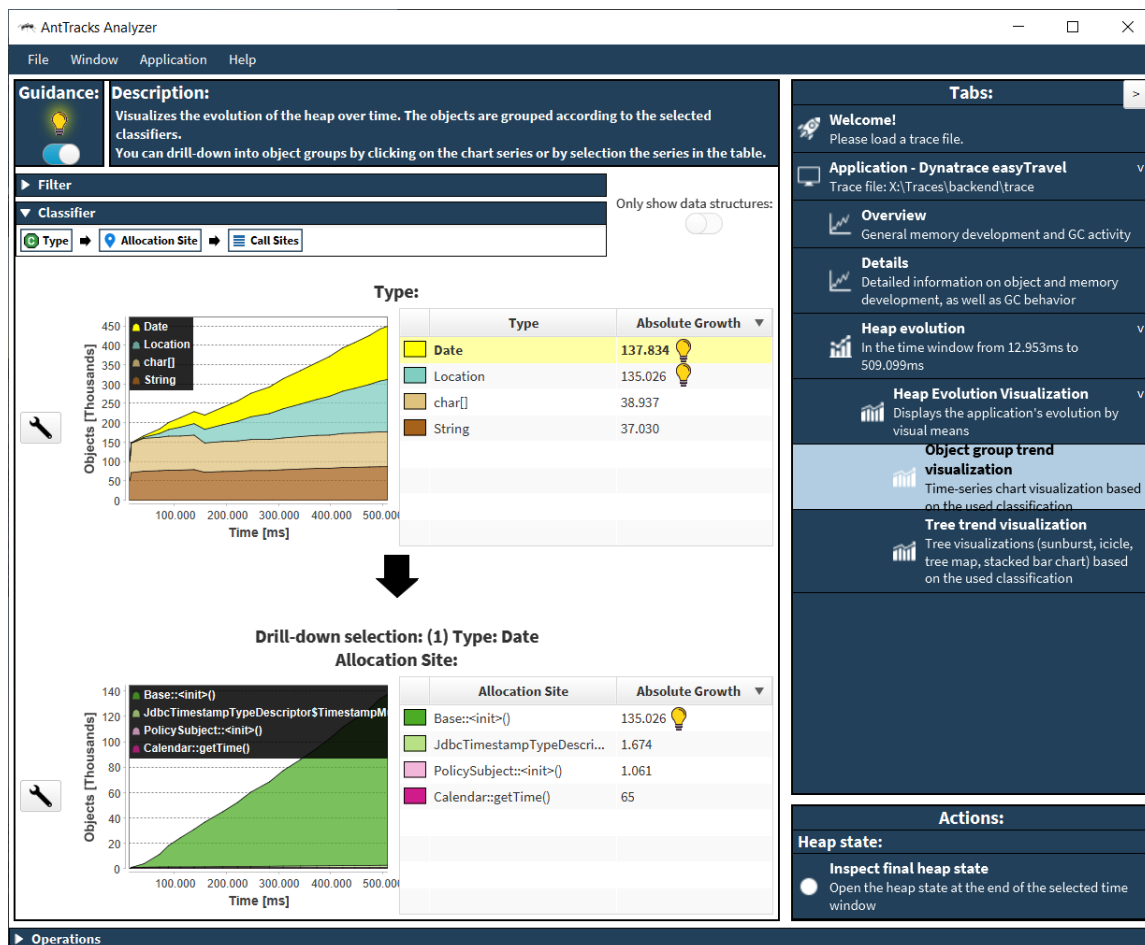


Fig. 10. The *TrendViz* view provides information on the heap evolution over time, i.e., which kinds of objects accumulated the most, including a drill-down feature to inspect suspicious object groups in more detail.

resulting memory trees is then visualized using time-series charts. Without guidance, users would have to select this list of classifiers on their own. Since GE aims to especially help novice users, we free them from this task by grouping the heap objects automatically by their types and their allocation sites. When opening the view, the evolution of the first level of the memory trees, i.e., the evolution of the objects grouped by type, is visualized, as shown on the top of [Figure 10](#) and in [Figure 8b](#). This is where GE comes into play.

Detection. We automatically detect the type of which the most objects accumulated over time. In easyTravel, the objects that accumulated the most are objects of type Date, as shown in [Figure 8b](#). Objects of this type are most probably involved in a possible memory leak. If multiple types exhibit similar strong growth (such as Location), all of them are suggested to the user for further inspection.

Highlighting. The chart series and the table entry of the suspicious type will be highlighted (yellow overlay) as shown in [Figure 8b](#).

Explanation. AntTracks’s explanatory text for the strongest growing type(s) reads “Over the selected time window, the number of <type> objects increased by <absolute growth>. This corresponds to <relative growth>% of the total heap growth and could be an indication of a memory leak.”

Suggestion. The view provides two suggestions: Either to stay on the view to drill-down to inspect where the suspicious objects have been allocated, or to go to the heap graph view to visually inspect the GC roots that keep the suspicious objects alive. Following the first suggestion opens a second time-series chart below the current one, which shows the evolution of the second level of the memory trees. For example, selecting the Date objects for drill-down opens a second chart that shows where the Date objects have been allocated over time, as shown on the bottom of [Figure 10](#). The same detection, highlighting and explanation steps as described above are then performed for the allocation sites, and users are suggested to visualize the objects of the strongest growing type that have been allocated at the allocation-heaviest allocation site in the heap graph view.

[Figure 8b](#) shows the memory evolution of easyTravel, grouped by type, on the AntTracks TrendViz view. Using the *Type classifier* as first grouping criterion was automatically performed by GE as we followed the suggestion on the overview. On the TrendViz view, AntTracks’s GE automatically *detected* that the objects that accumulated the most in easyTravel in the selected time window are those of type Date (*highlighted* in yellow) and Location. AntTracks’s GE *explains* that the Date objects are the major suspects for a possible memory leak since about 30% of the overall heap growth can be accounted to them. We then followed the *suggestion* to use the heap graph view to inspect the paths to the GC roots and thus find out which objects and GC roots (indirectly) keep the Date objects alive.

4.2.3 Heap Graph View.

Objects are kept alive because they are directly or indirectly reachable from GC roots. The heap graph view ([Figure 11](#)) is a newly introduced analysis view in AntTracks that is used to visually explore the references between heap objects and GC roots. This view was developed with GE support from the start to help users in detecting and understanding suspicious paths from objects to their GC roots, called *bottom-up analysis*. In the following, we will briefly explain the view’s interaction features before we present its guided exploration operations.

A heap may contain millions of objects, each of them referencing other objects. Thus, visualizing every object as a separate node and every reference as a separate edge is not feasible. Instead, our

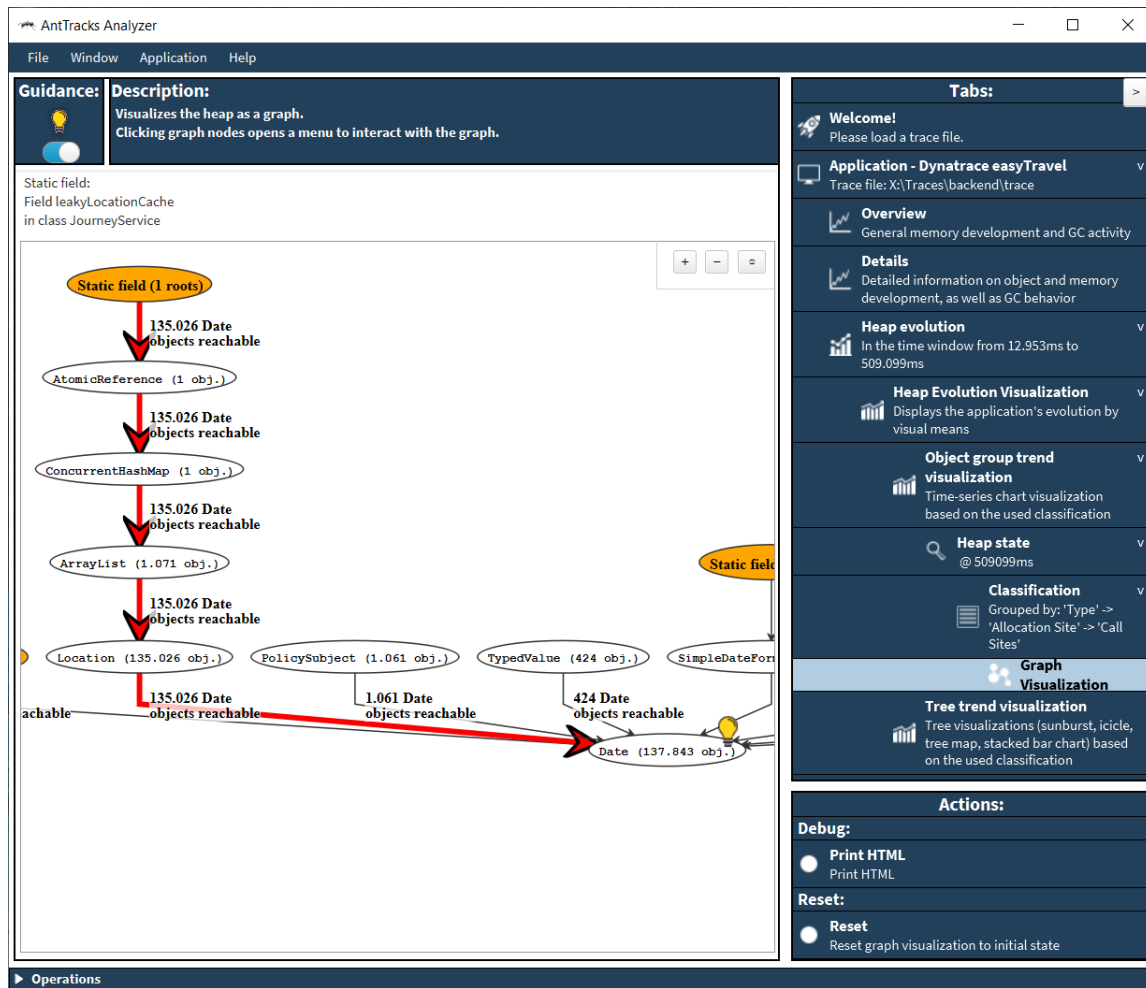


Fig. 11. The *Heap graph* view provides inspection features to analyze keep-alive relations between objects, paramount information to find the culprits of a possible memory leak.

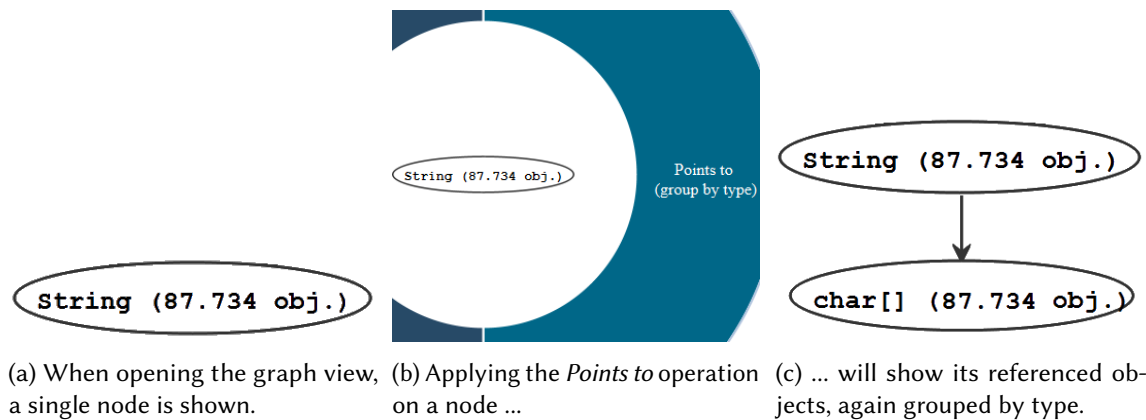


Fig. 12. Neighborhood analysis in the graph view.

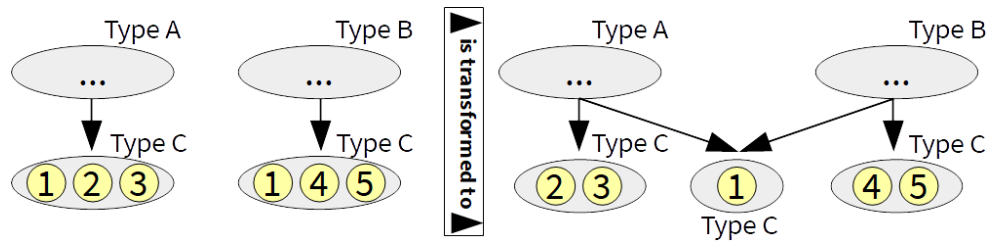


Fig. 13. The heap graph view groups objects by their types and extracts objects into separate nodes if they would be part of multiple nodes.

approach groups objects of the same type. When opening the heap graph view for objects of a specific type (e.g., inspecting all `String` objects), only a single node will be shown (for example 87.734 strings aggregated into a single node in Figure 12a). Users can explore the neighbors of a node by applying the *Points To* and *Pointed From* operations. These operations show the objects referenced by the node's objects or the objects referencing the node's objects respectively (again grouped into nodes based on their types). For example, applying the *Points To* operation (Figure 12b) on the `String` node will show all `char[]` objects referenced by the strings, as shown in Figure 12c.

Applying these neighborhood operations multiple times could lead to a situation in which one object is contained in two different nodes, which is illustrated on the left of Figure 13. The two top-most nodes represent objects of type A and type B, respectively. Assume that these objects point to objects of type C, namely the objects of type A point to the objects 1, 2 and 3, and the objects of type B point to the objects 1, 4 and 5. Since object 1 would be present in two different nodes, we extract it into a separate node, as shown on the right of Figure 13. Thus, after each operation that added nodes to the graph, we apply this technique on the whole graph to ensure that no heap object is contained in more than one graph node.

The view also supports operations that do not only involve the direct neighbors, but also operations to inspect *paths*. One of these operations is the *Paths to GC Roots* operation, which shows every path to GC roots starting from a selected node. Such a path represents a chain of objects that keep each other alive, starting at the object pointed by the GC root. While certain kinds of GC path inspection are also possible in other memory tools, nearly all these tools do not visualize these paths by graphical means but only in tree views. This has certain drawbacks. First, unwinding long paths in a tree view can be tedious. Second, it easily becomes confusing if multiple paths are shown. Third, tree views cannot display circular reference patterns. And lastly, most tools only support the inspection of GC root paths for a single object, not for object groups.

Since it can be rather complex to apply this view's features correctly and to interpret the resulting graph, AntTracks's guided exploration supports the user in multiple phases:

Detection #1. When the graph view is initially opened, a single node is shown, representing the heap objects for which we want to explore the paths to the GC roots.

Highlighting #1. This node is animated to draw attention to the fact that nodes are clickable.

Explanation #1. Since the user may have never investigated a memory problem before, AntTracks explains that the highlighted objects are kept alive because they are (indirectly) reachable from some GC roots that have yet to be explored.

Suggestion #1. To find out which GC roots keep the objects alive, we suggest to perform the *Paths to Most Interesting GC Roots* operation, an operation similar to the *Path to GC Roots* operations explained earlier. To create the paths to *all* roots, the *Pointed From* operation is automatically

applied multiple times, each time on the graph nodes that have been created in the previous step, until every path reaches a GC root. The *Paths to Most Interesting GC Roots*, instead of applying the *Pointed From* operation to every newly created node, applies this operation only to those nodes that reach at least 5% of the objects of the clicked node. For example, in [Figure 8c](#), the first *Pointed From* operation is applied to the Date node, which creates nodes for Location, PolicySubject, TypedValue and so on. While the *Path to GC Roots* algorithm would continue with all these nodes, the *Paths to Most Interesting GC Roots* algorithm only continues with the Location node. This is repeated until the graph cannot be expanded anymore, which results in the state shown in [Figure 8c](#).

Detection #2. Once the most important paths to the GC roots are shown, we automatically detect the path that reaches the most objects of the selected node, i.e., the path on which most objects may be kept alive.

Highlighting #2. The detected path is highlighted in red, and the thickness of the edges is adjusted according to the number of reachable objects, as shown in [Figure 8c](#).

Explanation #2. In the example from [Figure 8c](#), we explain that 135,026 Date objects are reachable from the leftmost path, while only 1,061 are reachable from the second-leftmost path. Consequently, we point out to the user that it is much more important to inspect the leftmost path than any other path. Once the most suspicious path is highlighted and its importance is explained, it is up to the user to try to “cut” the path somewhere. This cut must happen on the source code level by setting references that keep objects alive to null, or by removing objects from their containing data structures.

Suggestion #2. We currently suggest the user to start the source code inspection at the GC root and to traverse the references according to the types shown in the heap graph view. In future work, we will improve this suggestion step, for example by taking into account data structure boundaries or by including static source code analysis.

To inspect the root cause of easyTravel’s memory leak, we followed the *suggestion* to apply the *Paths to Most Interesting GC Roots* operation to find those GC roots that keep most of the Date objects alive. The result of this operation, including GE’s *highlighting*, can be seen in [Figure 8c](#). AntTracks’s guided exploration *explains* that many Date objects are alive because they are reachable from objects along the path highlighted in red. The current GE implementation in AntTracks verbalizes the problem in the following way: 135,026 Dates are kept alive by 135,026 Locations. These Locations are kept alive by 1,071 ArrayLists. These ArrayLists are kept alive by a single ConcurrentHashMap. This ConcurrentHashMap is kept alive by a single AtomicReference. Finally, this AtomicReference is kept alive because it is stored in a static field called locationCache in the class JourneyService. To reduce the number of Date objects, you have to cut this path somewhere. You can achieve this by setting references to null, or by removing objects from their containing data structures. Also check why the Date objects are added in the first place. Are they contained in the mentioned data structures on purpose?

With this information, the user should be able to locate the reported objects in the source code. In this example, we looked up the variable name locationCache and checked the variable’s usage. As the name suggests, the map serves as a cache, but its implementation was broken. There is a single line in the source code where new ArrayList<Location> objects are added if no matching key is already found in the cache. However, the class used as key in the ConcurrentHashMap neither implemented hashCode nor equals. Thus, every request (even for an already existing key) resulted in a cache miss and ultimately led to the problem that too many Location objects and Date objects

were created and kept alive. Implementing the two missing methods immediately resolved the problem.

5 GUIDED EXPLORATION OF MEMORY CHURN

To provide support for memory churn analysis, AntTracks encompasses a *short-living objects view* [90] that enables users to inspect those objects that are allocated in large quantities and die shortly afterwards. GE should help users to detect time windows that exhibit suspicious memory churn behavior as well as to guide them in finding those source code locations that should be inspected to reduce the churn.

5.1 Mapping of Memory Churn Analysis Process Steps to Views

The overall goal of memory churn analysis is to reduce the number of allocations happening in *memory churn hotspots*. Figure 14 shows a simplified model of such a memory churn analysis process. The first step is to detect a time window that covers a memory churn hotspot, i.e., a time window with strongly fluctuating memory utilization. The user then has to find out which types of objects are responsible for the churn and where these objects have been allocated. These locations can then be inspected in the source code to fix the problem. In AntTracks, these tasks are performed on two different views:

- (1) The *Details view* plots a detailed evolution of the memory footprint, where certain patterns indicate churn.
- (2) The *Short-living objects view* drills down into suspicious object groups to extract their types and allocation sites.

Both mentioned views already existed in AntTracks and have been extended with GE support as part of this work.

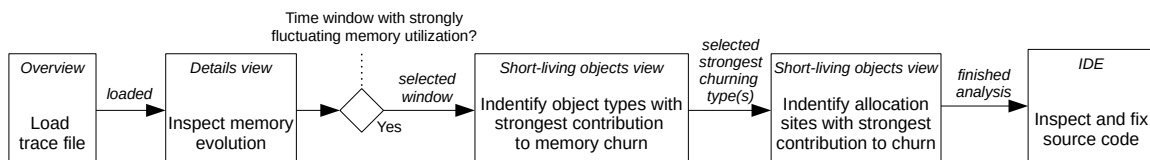


Fig. 14. Simplified task model of the typical steps performed during memory churn analysis, mapped to their corresponding AntTracks views.

5.2 Guided Exploration Support Operations for Memory Churn Analysis

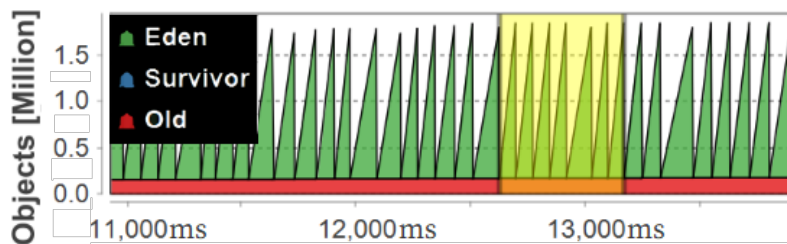
In this section, we show how GE is now supported in the two views identified in Section 5.1. We explain the views' general features and their newly supported GE support operations *Detection*, *Highlighting*, *Explanation*, and *Suggestion*.

To showcase how GE in AntTracks now supports users in analyzing and fixing memory churn, we present how the newly introduced guidance features have been used to inspect a benchmark of the *Renaissance benchmark suite* [61]. This suite is composed of modern, real-world, concurrent, and object-oriented workloads. Since this benchmark suite is rather new, it has not yet been the subject of a detailed memory study [42]. Thus, it is perfectly suited to test whether AntTracks's GE is able to guide users to the root cause of memory problems in applications even unknown to the inspector. We downloaded the benchmark suite in version 0.9, created a trace file of every benchmark and loaded these trace files into AntTracks and inspected the memory churn time windows that were automatically detected by GE. One benchmark that attracted our attention was

finagle-http. According to the benchmark’s documentation, it *sends many small Finagle HTTP requests to a Finagle HTTP server and awaits response*. All automatically detected and highlighted problem patterns in this application are shown in Figure 15 and will be explained in detail in the following.

5.2.1 Details View.

The details view (Figure 16) plots the memory consumption GC-wise, i.e., the view’s plots contain one data point at the beginning of a garbage collection (high memory consumption) and one at the end (low memory consumption). Thus, every garbage collection appears as a spike. When users investigate this view without guidance, they have to know that their task is to detect a time window



(a) Automatically detected memory churn hotspot in the *finagle-http* benchmark.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
4 GCs survived	7,686
1 GCs survived	21

(b) The guidance on the short-living objects reports that nearly all died objects did so without surviving a single garbage collection.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
Promise\$WaitQueue\$\$anon\$4	2,494,576
Promise\$Monitored	2,494,393
Future\$\$anonfun\$onSuccess\$1	2,494,362
FinagleHttp\$\$anonfun\$runIteration\$1\$\$anon\$2	2,494,361
char[]	3,196

(c) Inspecting the types of the frequently dying objects reveals four types (that are automatically detected) that seem suspicious.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
...	
FinagleHttp\$\$anonfun\$runIteration\$1\$\$anon\$2	2,494,361
FinagleHttp\$\$anonfun\$runIteration\$1\$\$anon\$2	2,494,361

(d) Inspecting the allocation sites of these frequently dying objects leads to the source code locations that have to be checked.

Fig. 15. Memory churn analysis in AntTracks.

that contains high and frequent spikes. Yet, observing and interviewing memory monitoring tool users revealed that especially novices are often not aware of other memory problems beside memory leaks. They often lacked background knowledge to recognize high memory churn patterns as suspicious and worthy of inspection, a reason why we try to ease memory churn detection and analysis using guided exploration through the following support operations.

Detection. We apply the automatic memory churn time window detection algorithm by Weninger et al. [89] to detect memory churn hotspots.

Highlighting. Detected memory churn hotspots are highlighted with a yellow overlay (see Figure 15a), similar to memory leak time windows, following the HCI principle *consistency* [6].

Explanation. We explain the term *memory churn*, since most novice users may not be familiar with it: *AntTracks detected a time window where your application throws away over <garbage> MB per second, which is called high memory churn. This occurs when many short-living objects are being allocated in a short time span, leading to frequent garbage collections. Please note that too many GCs can slow down your application even if the GCs themselves are very quick.*

Suggestion. Our suggestion to the user is to use the *short-living objects* view to find out which objects cause the memory churn and where these objects have been allocated.

The mentioned memory churn hotspot detection algorithm is automatically run by AntTracks's GE every time a trace file is loaded. If such a hotspot is detected, as it was the case for the *finagle-http* benchmark, AntTracks suggests the user to switch to the details view to visualize it. Figure 15a shows the automatically detected memory churn hotspot in the *finagle-http* benchmark, for which AntTracks explains that about 500 MB are allocated and freed every second within the highlighted

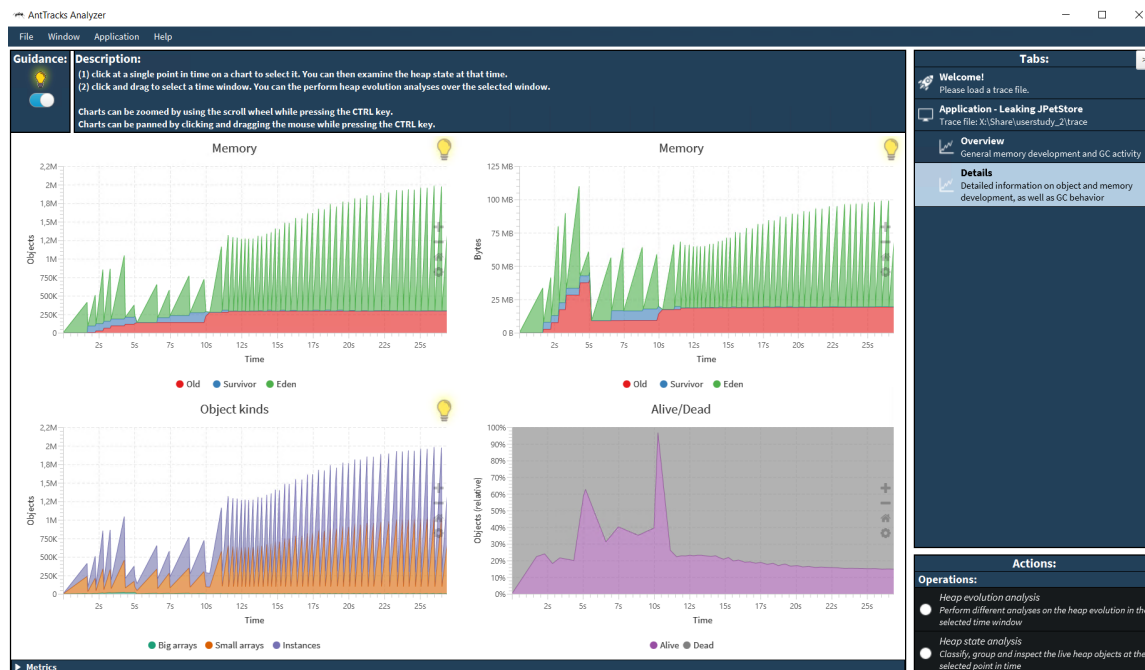


Fig. 16. The *Details* view provides more detailed information about the memory behavior, enabling the detection of spike patterns that hint at memory churn.

time window. Based on this, it suggests to explore the objects that make up this garbage in more detail on the short-living objects view.

5.2.2 Short-living Objects View.

The short-living objects view (Figure 17) calculates the *age* of each object that died within a selected time window. It uses this information to guide users to those objects that die shortly after their allocation, which are the major reason for memory churn. We define the age of a heap object as *the number of garbage collections it survived*. Even though more detailed death time algorithms exist, AntTracks uses this simple age definition as it can be reconstructed during trace parsing without additional overhead [90]. For example, the *Merlin* algorithm [25] used by *Elephant Tracks* [68, 69] can calculate more exact object death times, yet it causes a several 100-fold increase in the analyzed application's run time [101].

The short-living objects view in AntTracks introduces a new classifier: the *Age classifier*. As shown in Figure 15b, applied on a died object, the age classifier returns the string “<x> GCs survived” as its classification. Like most views in AntTracks, the short-living objects view uses a tree table view to display the objects that were freed by the garbage collector. By default, these died objects are grouped first by age, then by type, and then by allocation site. They are sorted based on the number of objects that have been collected by the GC in the selected time window.

To make it easier for novices to learn and interpret this view, GE automatically detects suspicious objects and presents them to the user in multiple steps.

Detection #1. As the first classifier applied on the died objects is the *Age classifier*, we automatically detect how many objects died without surviving a single garbage collection.

Highlighting #1. Figure 15b shows how we highlight objects in the tree table view by assigning a special background color to the respective rows.

Name	Collected objects	Collected memory
Overall	12.191.082	579,1 MB
0 GCs survived	12.119.646	576 MB
char[]	5.035.395	313 MB
String	2.170.537	52,1 MB
String::toString()	2.026.445	48,6 MB
(hidden internal call sites)	~1.951.241	~46,8 MB
CatalogService::getCategoryList()	~1.943.042	~46,6 MB
CatalogService::getProductListByCategory()	~8.198	~196,8 KB
HtmlUtil::encode()	~13.709	~329 KB
UrlBuilder::build()	~9.788	~234,9 KB
ObjectInputStream\$BlockDataInputStream::readU	~9.756	~234,2 KB

Fig. 17. The *Short-living objects view* helps the user to drill-down into object groups that contributed the most to the memory churn.

Explanation #1. A text explains that allocating large numbers of objects only to discard them shortly afterwards greatly increases the GC frequency which impacts the performance, followed by the info that $x\%$ of the objects that died in the selected time window did not even survive a single garbage collection. The text also explains that especially objects that do not even survive a single garbage collection are the main reason for memory churn and that they should be investigated.

Suggestion #1. As a next step, we suggest the user to expand the *0 GCs survived* row to check the types of the objects that died without surviving a single garbage collection.

Detection #2. GE detects those types that caused the most garbage within the selected time window. We empirically tested this view on various applications and determined that those types that account for at least 10% of the garbage should be highlighted. If no type accounts for at least 10% of the garbage, the type of which the most objects died is selected.

Highlighting #2. [Figure 15c](#) shows an example where four types are highlighted since each of them makes up about 25% of the overall garbage.

Explanation #2. Following text informs the users about suspicious types: $\langle x \rangle$ types have been detected as the major suspects for memory churn: $\langle \text{list of types} \rangle$. They account for $\langle y \rangle\%$ of all objects that died without surviving a single garbage collection.

Suggestion #2. For each suspicious type, GE suggests to inspect its allocation sites by expanding the type's row.

Detection #3. Among all allocation sites, the one at which the most objects were allocated is detected. According to our experience, most memory churn hotspots are caused by a single allocation site.

Highlighting #3. Again, the respective tree row is highlighted, as shown in [Figure 15d](#).

Explanation #3. AntTracks explains that an allocation site is the location in the code at which an object has been created, and that allocation sites where many short-living objects are created should be inspected in the source code. We further provide hints on what typical root causes of memory churn might look like. For example, allocations inside heavily-executed loops are dangerous. Another typical mistake is the careless adding and removing of boxed primitives to data structures, e.g., `ArrayList<Integer>`. Every time a primitive is added to such a data structure it is wrapped into a heap object, which can cause unnecessary memory overhead. One last example is the careless use of streams. Typical mistakes are (1) to perform multiple map operations that unnecessarily create many short-living intermediate objects, or (2) to use map when working with primitives instead of using the respective memory-efficient mapping operation such as `mapToInt`, or (3) to use filter operations too late in the chain of operations, leading to unnecessary operations and allocations to be performed.

Suggestion #3. We suggest to review the source code with regard to whether the executed allocations are really necessary. To reduce the number of allocations, existing objects could be reused [29, 47], for example by implementing a caching strategy and/or by using design patterns such as the *prototype pattern* or the *flyweight pattern* [20]. Future work encompasses to statically inspect the suspicious allocation site to derive context information about the allocations and to give more precise suggestions to the user.

When investigating the *finagle-http* benchmark, the first thing that is highlighted on this view is that over 99.9% of the objects that died in the selected time window (10,012,077 out of 10,019,784)

```

1 val response: Future[http.Response] = client(request)
2 for (i <- 0 until NUM_REQUESTS) {
3   Await.result(response.onSuccess { rep: http.Response =>
4     totalLength += rep.content.length
5   })
6 }

```

Listing 1. Problematic part of `FinagleHttp.runIteration()`.

```

1 val response: Future[http.Response] = client(request)
2 val h = { rep: http.Response => totalLength += rep.content.length }
3 for (i <- 0 until NUM_REQUESTS) Await.result(response.onSuccess(h))

```

Listing 2. Fixed version of `FinagleHttp.runIteration()`.

did not even survive a single garbage collection, as shown on [Figure 15b](#). Following the suggestion to inspect the types of the died objects (shown in [Figure 15c](#)) reveals that most of the died objects are divided almost equally among four types. It may be worth to mention that `finagle-http` is a Scala application, which typically produces longer type names than Java. For each of the four types, GE suggests to expand the respective row and to inspect the allocation sites of the different types. In this case, all objects of a given type that died in the selected time window were allocated at a single allocation site. The allocation sites of the first three types are within library methods which we cannot modify. Yet, the fourth type’s allocation site is located in the `FinagleHttp` class, the benchmark’s main class (see [Figure 15d](#)). Since Scala type names and allocation sites can be quite hard to read, we integrated rudimentary support into AntTracks’s GE to translate them. For example, in the explanation text it translates `FinagleHttp$$anonfun$runIteration$1$$...` to *anonymous Scala function objects that have been allocated in method `runIteration` of the class `FinagleHttp`*.

Since such a rapid allocation and collection of anonymous function objects is unlikely to be intentional, we looked up the method’s source code. [Listing 1](#) shows the problematic part. In the loop, a large number of anonymous function objects are created that wait for an HTTP request to succeed before incrementing the counter `totalLength`. [Listing 2](#) shows our fix for this problem. Only a single response handler is created which is reused for every HTTP request. This fix reduced the overall amount of allocated temporary objects by about 25% and sped up the application by about 5%.

6 PRELIMINARY USER FEEDBACK

Even though a detailed user study is still missing (but planned as future work, see [Section 8](#)), we wanted to gather preliminary user feedback to get a general idea of how AntTracks’s new GE features may help novice users. To this end, we asked three PhD students and two master students that work as assistants at our institute¹ to use AntTracks and its new guidance features. All of them have a background in computer science and software engineering, and the participants reported experience in software development ranging from four to eight years. None of them had used AntTracks before and all of them stated that they had no background in memory analysis (i.e., they classified themselves as *novices* with regard to memory analysis). In separate sessions, each participant was given a memory leak and a memory churn analysis task, both of which were taken from a user study on the usability of memory monitoring tools [91]. We asked the participant to ‘think aloud’ [27, 31, 55], i.e., to describe what they are doing, to comment on any of their

¹None of them is involved in the development of AntTracks or this research.

concerns, and to say whatever comes to their mind while solving the given tasks. This way, we were able to collect a number of interesting *observations* and *think-aloud statements*, which we used in combination with feedback collected during a short final *interview* to initially assess AntTracks's GE system.

6.1 Study System

We selected the web application JPetStore 6 [52] as our study system. JPetStore has been widely used in research projects [17, 32, 35, 36, 83]. It models a minimalistic web shop for pets and uses a clearly structured class hierarchy. We chose JPetStore since its straightforward structure can be expressed well in a simple UML class diagram [7]. This UML diagram was handed out at the beginning of each session, which made it easy for the participants to comprehend the system's structure without being familiar with its source code. This helped to mitigate the risk of participants not finishing the study tasks [92]. To prepare the system for the study, we modified the JPetStore source code to contain two memory anomalies. We seeded the system with a memory leak by keeping shop item objects alive after their web page has been requested and a memory churn hotspot by using a Java stream inefficiently to process database responses.

We created AntTracks trace files before the user study for both the memory leak and the memory churn problem. In particular, we simulated heavy load by sending numerous requests to the different web pages of the application.

6.2 Tasks

The participants were given the trace files and had to complete the following five tasks. On each view, they were allowed to use AntTracks's GE features to receive guidance by the tool:

- *Memory leak detection*, i.e., they had to recognize and correctly classify a suspicious memory growth time window as such.
- *Trend analysis*, i.e., the participants had to find out which kinds of objects accumulated the most over this window.
- *Graph-based GC root analysis*, i.e., on the graph view, the participants had to find out which GC roots cause the memory leak. They were then shown the source code to try to fix the problem.
- *Memory churn detection*, i.e., after (hopefully) fixing the memory leak, the participants had to recognize and correctly classify a memory churn hotspot (i.e., a frequent spike pattern in the memory chart).
- *Short-living objects analysis*, i.e., they had to collect information (such as object types and allocation sites) about the churning objects. They were then again asked to locate and fix the memory problem in the source code based on their findings.

6.3 Feedback

All users were able to find the problematic source code locations relevant to the memory problems and actively expressed that they liked the guidance system (“*The light bulbs were great if you got lost or did not know what to do*”). They stated that the analysis flow and the hints during the memory leak analysis (cf. Section 4) as well as the memory churn analysis (cf. Section 5) are well chosen (“*The number and order of steps seemed natural to me*”). All users agreed that the amount of text shown in the individual hints is reasonable and not overwhelming. Despite their length, the explanations of common root causes for an observed problem and how to fix them (e.g., a list of possible root causes for high memory churn) have been praised as being very helpful to novices

(“*Experienced or power users may not need them, but they were great help for me as a beginner*”). One user also positively highlighted the use of formatting in the information text.

Nevertheless, some participants also reported that, even though the guidance helped them to *find* the problematic source code locations, they struggled to *fix* the problem (“*I know that the problem is due to a lot of Strings and Products being allocated here, but I cannot find out how to fix it; I think it has something to do with this stream*”). This suggests that future work should explore how we can further support users after their final analysis step in the monitoring tool, for example by adding guidance features to the IDE.

One improvement we already implemented based on this preliminary feedback concerns the way how we present available hints. In our initial version, each view had a single light bulb in its top-left corner that stored all the hints for this view. If new hints became available, e.g., after performing a certain action, they would be added to that light bulb’s list of hints. The users expressed that they would rather have a separate light bulb for every hint that is placed next to the UI element it refers to. This makes it more clear when a new hint becomes available, as a new light bulb appears.

As a final question, we asked the users whether they think that they would have been able to use AntTracks to find the root causes of the problems without guidance. All of them said that they think they would have eventually succeeded using a trial-and-error approach. However, all of them also stated that they are certain that it would have taken them far more time to complete the tasks.

7 GUIDED EXPLORATION IN ANOTHER DOMAIN: THREAD LOCK CONTENTION MONITORING

While the main focus of this work was to show how GE can be integrated into an interactive memory monitoring tool, we are confident that the general GE method presented in [Section 3](#) can also be useful for monitoring tools of other domains. While more work and research still has to be performed in this direction, we initially asked authors² of an interactive thread lock contention monitoring tool [26, 72] on their opinion whether they think if GE could also be integrated into their tool.

In their positive response, they outlined how they would proceed to integrate guided exploration. They explained the typical analysis process for lock contention, all its involved steps and outlined how these steps map to views in their tool. Even though GE has not yet been integrated into their tool, we will outline their detailed response on how they could use GE to guide users on their search for thread locking problems in their applications.

7.1 Mapping of Thread Lock Contention Analysis Process Steps to Views

According to the tool authors, users who use their tool for the first time are mostly interested in (1) which shared resource (monitor) is blocking threads the most, (2) which method spends the most time waiting for the resource and (3) which method holds the resource the most and thus causes the most waiting time. This well-defined default analysis flow is also visualized in [Figure 18](#). The analysis steps for all these tasks happen on the same view of their tool, its *drill-down view*.

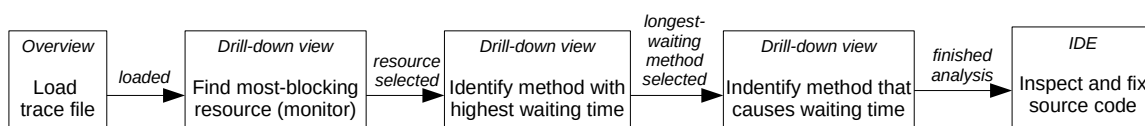


Fig. 18. Simplified task model of the typical steps performed during thread lock contention analysis, mapped to views in the monitoring tool.

²None of them is involved in the development of AntTracks or this research.

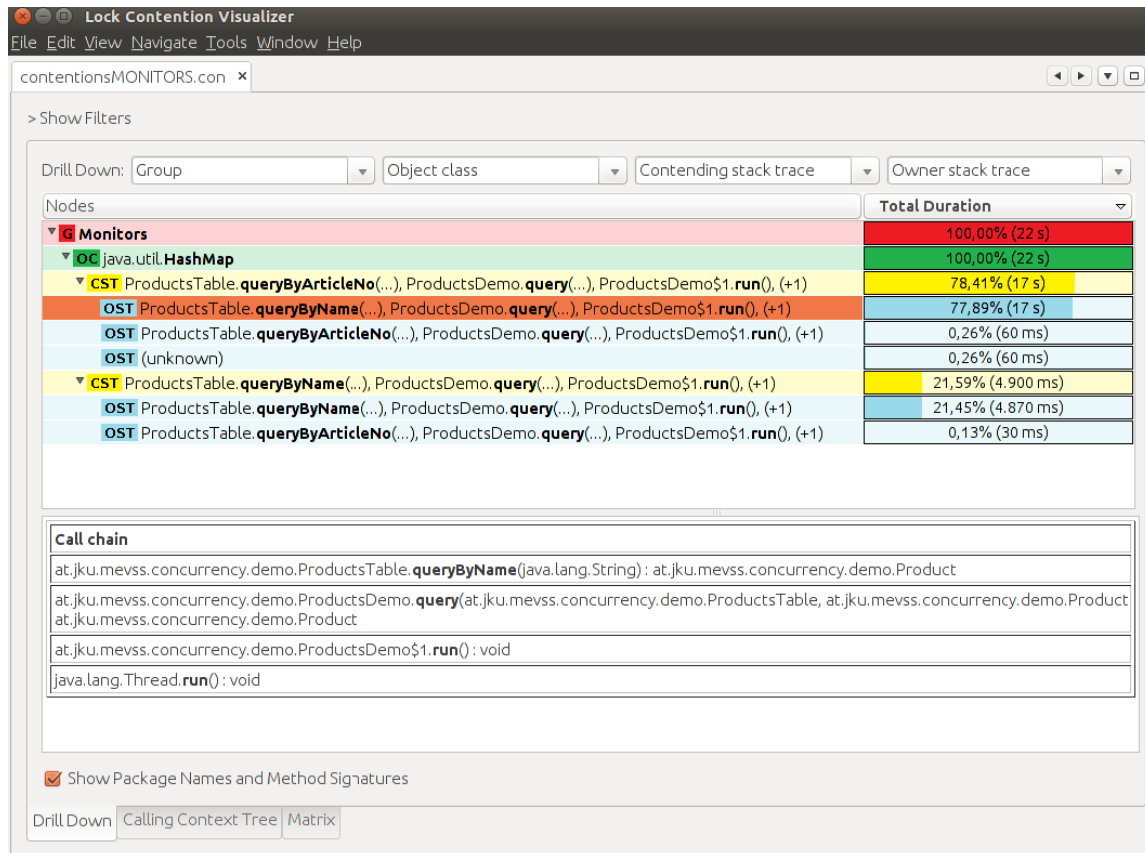


Fig. 19. The drill-down view in this thread lock contention monitoring tool could also be extended with guided exploration features (slightly modified figure taken with permission from [74] - Figure 4.4).

7.2 Guided Exploration Support Operations for Thread Lock Contention Analysis

The tool authors replied that they think that users could be guided well through the mentioned analysis steps and their tool's drill-down view (Figure 19) using on the four GE support operations detection, highlighting, explanation and suggestion. They also mentioned that they think that a "lightbulb-based" info mechanism to inform users about possible guidances, similar to the one used in AntTracks, could be easily integrated. In the following, we report their ideas on how to integrate GE's guidance operations into their tool.

Detection #1. Once a trace file (containing information about lock contentions that happened in the monitored application) has been loaded, the user can select criteria based on which the drill-down view groups the data. By default, the data is automatically grouped to best support the default task given in Figure 18. In this grouping, the first level of the tree splits all lock contentions that happened based on the resources that caused them. Following our GE method, the authors would improve their tool to automatically detect the resource that caused the most waiting time.

Highlighting #1. Since the majority of their UI is also structured in a table view (see Figure 19), highlighting could work in a similar way to how it is performed in AntTracks, i.e., by using colored overlays.

Explanation #1. They would explain to the user that lock contention happens when a thread T2 wants to lock a resource R, but that resource is already locked by another thread T1. T2 thus has to wait until R is released by T1, an undesired behavior in multi-threaded applications. Their tool would continue to explain that it just detected the resource that caused the most waiting time, and that the locks involving this resource should be inspected in more detail.

Suggestion #1. They would inform the user that they should expand the respective tree row to inspect *where* the most threads are waiting for the shared resource.

Detection #2. The second tree level groups all lock contentions involving a given shared resource by the methods in which threads had to wait for the resource. The tool could automatically detect the method where threads had to wait the most.

Highlighting #2. Again, the respective tree view row could be highlighted with an overlay.

Explanation #2. The tool authors would first introduce certain terminology such as *contending method* or *contending stack trace*. Following, they would explain that the highlighted row shows the method that had to wait the most for the most-blocking shared resource.

Suggestion #2. Since we just found the method that had to wait the most, the last vital information is to find out where the lock was held the most during this waiting time. Thus, the tool would suggest to step one level deeper into the tree to gather this information.

Detection #3. On the final tree level, the tool would automatically search for the method that caused the most waiting time by holding the respective shared resource while another thread wanted to obtain it.

Highlighting #3. Again, the respective tree view row could be highlighted with an overlay.

Explanation #3. After the final piece of information was collected, the explanation would summarize *which* object caused the most waiting time, *where* threads were waiting the most for this object, and *where* the object was mostly held while others were waiting.

Suggestion #3. To investigate the problem in the source code, the tool would suggest to look up both mentioned methods. Most locks in Java are caused by requests to shared resources in synchronized blocks in the form of `synchronized(sharedResource) { . . . }` or by operations performed by classes of the `java.util.concurrent` package. It is the developer's task to ensure that these locked regions span as few operations as possible, and that locking is only performed where needed.

Even though further evaluation is needed, this detailed description by the lock contention monitoring tool authors fosters our belief that the GE method can also be useful to monitoring tool developers of various monitoring domains.

8 CURRENT LIMITATIONS AND FUTURE WORK

In this section, we discuss current limitations of the general GE approach, GE in the memory monitoring tool AntTracks, and how we plan to tackle these limitations in future work.

8.1 Guided Exploration in AntTracks

The main focus of this work was to explore how novice users could be better guided in interactive memory monitoring tools such as AntTracks. While the reactions during a preliminary user

feedback (see [Section 6](#)) were promisingly positive, our approach still has limitations that should be investigated in the future.

User Study. Based on the preliminary feedback we collected, we strongly believe that GE in AntTracks makes it easier to use and learn the tool, especially for novice users without expertise in memory monitoring. We presented user scenarios to demonstrate the usefulness of GE and to showcase how the guidance supports users when inspecting applications. Nevertheless, a more thorough evaluation is still missing. We thus plan to conduct a user study to compare the performance of participants who use AntTracks’s GE support with the performance of those who try to resolve memory problems without guidance. It would also be interesting to check whether GE is helpful to both novice users as well as advanced users, or if advanced users prefer to use the tool without guidance.

Guided Exploration Integration in IDEs. Currently, there is a clear separation between the guided analyses in AntTracks and unguided source code inspection in the IDE. For example, after users were guided to a suspicious allocation site in AntTracks, they still have to fix the source code in their IDE without further guidance. Developing an IDE plugin [[3](#), [11](#)] and using hybrid static and dynamic analysis [[15](#)] would allow us to highlight suspicious code segments in the IDE, continuing GE on the source code level.

Heap Graph Visualization. In [Section 4](#), we presented AntTracks’s new heap graph view to inspect a heap state in a visual way. This feature is still under development and evolves constantly. In the future, we plan to report in more detail on this new visualization technique, how it compares to other techniques for heap visualization [[59](#), [66](#), [95–98](#)], and how its guidance can be further extended.

8.2 Guided Exploration in General

In general, our GE method presented in [Section 3](#) is not restricted to the domain of memory monitoring. Yet, while its core idea could also be useful to monitoring tools of other domains, future work still has to be performed to evaluate the approach’s general applicability across domains.

Generalization. To mitigate the generalizability problem of our approach, we asked other monitoring tool developers for their opinion regarding the feasibility of implementing GE in their tool and if they think that their users would profit from it, as shown in [Section 7](#). Despite their positive feedback and a detailed explanation on how they would integrate GE into their tool, more monitoring tools from different domains should be inspected for possible GE support in the future.

Furthermore, the GE process outlined in [Section 3](#) currently serves more as an overview of the four guidance operations (detection, highlighting, explanation and suggestion). Yet, we did not discuss in detail how to implement them, for example based on certain characteristics the monitoring tool exhibits. Rabiser et al. [[62](#), [63](#)] already explored various kinds of characteristics based on which different monitoring tools could be classified and compared, even across different domains. We think that it may be possible to link certain characteristics of a tool to suggested ways of how to implement the different guidance operations, for example different ways of problem detection or highlighting. Such relations between tool characteristics and possible guidance operation implementations could be explored and discussed in more detail in the future.

Guided Exploration Tool Integration. In this work, we presented how GE has been integrated into AntTracks, using clickable lightbulb icons near UI elements for which guidances exist. Yet, we are certain that there are other possibilities on how to visualize guidances. To explore these possibilities, for example, one could organize a workshop where the participants should inspect

existing monitoring tools and discuss commonalities across them. Separated into groups, they could work out GE prototypes, i.e., how they would integrate guidance into these tools, and comment on each others ideas. These discussions could lead to a more detailed description and understanding of the GE approach in the future, as well as to better ways on how to visually support it.

Rule-based Guidance Definition. AntTracks uses a consistent way to display *explanations* and *suggestions* throughout its various views. Unfortunately, the code to *detect* suspicious information as well as to *highlight* the respective UI region is currently hard-coded within every view.

In the future, instead of modifying the underlying source code, we would like to be able to define rules for guided exploration externally using a domain specific language that allows definitions such as “*If pattern X is detected: highlight UI element Y, show explanation text Z, suggest steps A and B*”. On the one hand, this poses various challenges such as how to access and abstract the data used by the view or how to specify (customized) UI element highlighting. On the other hand, it would make it much easier to integrate guided exploration into other tools besides AntTracks.

9 CONCLUSIONS

In this work, we presented *guided exploration*, a method that can be integrated into interactive monitoring tools in order to improve their learnability and usability. The goal of guided exploration is to support novice users, i.e., users who may lack the experience to recognize and analyze program behavior anomalies on their own. Guided exploration makes a tool easier to use by *guiding* users through the analysis process and helping them to *explore* the collected data until the root cause of a problem is found.

In general, guided exploration is an iteration of four support operations performed by a tool. According to GE, a tool should automatically (1) *detect* the most interesting piece of information in the current view, (2) *highlight* the UI elements where this information can be found, (3) *explain* the required background knowledge and the rationale why the highlighted information is important, and (4) *suggest* further analysis steps based on these findings.

In this work, we focused on guided exploration in interactive memory monitoring tools. We integrated our guidance approach into the memory monitoring tool AntTracks, namely for the processes of memory leak analysis and memory churn analysis. For both analyses, we explained in detail how the four support operations of guided exploration have been implemented. To demonstrate their applicability, we presented two user scenarios where two applications have been analyzed by following the explanations and suggestions of AntTracks’s new guided exploration system.

We hope that guided exploration can be of help to researchers and developers of interactive (memory) monitoring tools to better structure their analysis processes, making them more accessible to novice users.

ACKNOWLEDGMENT

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

REFERENCES

- [1] Alain Abran, Adel Khelifi, Witold Suryn, and Ahmed Seffah. 2003. Usability Meanings and Interpretations in ISO Standards. *Software Quality Journal* 11, 4 (2003), 325–338. <https://doi.org/10.1023/A:1025869312943>
- [2] Tarek M. Ahmed, Cor-Paul Bezemer, Tse-Hsun Chen, Ahmed E. Hassan, and Weiyi Shang. 2016. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web

- Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*. 1–12. <https://doi.org/10.1145/2901739.2901774>
- [3] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking Sketches and Diagrams to Source Code Artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symp. on Foundations of Software Engineering (FSE)*. 743–746. <https://doi.org/10.1145/2635868.2661672>
- [4] André Bauer, Marwin Züfle, Johannes Grohmann, Norbert Schmitt, Nikolas Herbst, and Samuel Kounev. 2020. An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 48–55. <https://doi.org/10.1145/3358960.3379123>
- [5] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proceedings of the Principles and Practices of Programming on The Java Platform (PPPJ)*. 76–89. <https://doi.org/10.1145/2807426.2807433>
- [6] Alan Blackwell and Thomas Green. 2003. CHAPTER 5 - Notational Systems - The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*. Morgan Kaufmann, 103 – 133. <https://doi.org/10.1016/B978-155860808-5/50005-8>
- [7] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. 2005. *The Unified Modeling Language User Guide - Covers UML 2.0 (Second Edition)*. Addison-Wesley.
- [8] Adriana E. Chis. 2008. Automatic Detection of Memory Anti-Patterns. In *Comp. to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 925–926. <https://doi.org/10.1145/1449814.1449911>
- [9] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O’Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of Memory Inefficiency. In *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP)*. 383–407. https://doi.org/10.1007/978-3-642-22655-7_18
- [10] Maria Christakis and Christian Bird. 2016. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 332–343. <https://doi.org/10.1145/2970276.2970347>
- [11] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genç Mazlami, and Harald C. Gall. 2018. PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE. In *Comp. of the 40th International Conference on Software Engineering (ICSE)*. 41–44. <https://doi.org/10.1145/3183440.3183481>
- [12] D. Christopher Dryer. 1997. Wizards, Guides, and Beyond: Rational and Empirical Methods for Selecting Optimal Intelligent User Interface Agents. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces (IUI)*. 265–268. <https://doi.org/10.1145/238218.238347>
- [13] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. 2003. Dynamic Metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 149–168. <https://doi.org/10.1145/949305.949320>
- [14] Dynatrace. 2017. *Demo Applications: easyTravel*. <https://community.dynatrace.com/community/display/DL/Demo+Applications++easyTravel>
- [15] Michael D. Ernst. 2003. Static and Dynamic Analysis: Synergy and Duality. In *Workshop on Dynamic Analysis (WODA)*. 24–27. <https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf>
- [16] Alexander Felfernig, Gerald Ninaus, Harald Grabner, Florian Reinfrank, Leopold Weninger, Dennis Pagano, and Walid Maalej. 2013. An Overview of Recommender Systems in Requirements Engineering. In *Managing Requirements Knowledge*. 315–332. https://doi.org/10.1007/978-3-642-34419-0_14
- [17] Florian Fittkau, Phil Stelzer, and Wilhelm Hasselbring. 2014. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *Proceedings of the Workshops & Tool Demos Track of the European Conference on Software Architecture (ECSAW)*. 28:1–28:4. <https://doi.org/10.1145/2642803.2642831>
- [18] Eelke Folmer and Jan Bosch. 2003. Usability Patterns in Software Architecture. In *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI)*. 93–97. <https://doi.org/10.1109/DSAA.2018.00057>
- [19] Tak-Chung Fu. 2011. A Review on Time Series Data Mining. *Eng. Appl. Artif. Intell.* 24, 1 (2011), 164–181. <https://doi.org/10.1016/j.engappai.2010.09.007>
- [20] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. 1993. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP)*. 406–431. https://doi.org/10.1007/3-540-47910-4_21
- [21] Josefina Guerrero García, Jean Vanderdonckt, and Christophe Lemaigre. 2008. Identification Criteria in Task Modeling. In *Proceedings of the 1st TC 13 IFIP Human-Computer Interaction Symposium (HCIS)*, Vol. 272. 7–20. https://doi.org/10.1007/978-0-387-09678-0_2
- [22] Bernhard Göschlberger and Peter A. Bruck. 2017. Gamification in Mobile and Workplace Integrated Microlearning. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services*

- (iiWAS) (Salzburg, Austria). 545–552. <https://doi.org/10.1145/3151759.3151795>
- [23] Juho Hamari, Jonna Koivisto, and Harri Sarsa. 2014. Does Gamification Work? - A Literature Review of Empirical Studies on Gamification. In *Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS)*. 3025–3034. <https://doi.org/10.1109/HICSS.2014.377>
- [24] William E. Hefley and Dianne Murray. 1993. Intelligent User Interfaces. In *Proceedings of the 1st International Conference on Intelligent User Interfaces (IUI)*. 3–10. <https://doi.org/10.1145/169891.169892>
- [25] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. 2006. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 476–516. <https://doi.org/10.1145/1133651.1133654>
- [26] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. 2016. Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 263–274. <https://doi.org/10.1145/2851553.2851559>
- [27] Andreas Holzinger. 2005. Usability Engineering Methods for Software Developers. *Commun. ACM* 48, 1 (2005), 71–74. <https://doi.org/10.1145/1039539.1039541>
- [28] Michal Hucko, Ladislav Gazo, Peter Simún, Matej Valky, Róbert Móro, Jakub Simko, and Mária Bielíková. 2019. YesElf: Personalized Onboarding for Web Applications. In *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization (UMAP)*. 39–44. <https://doi.org/10.1145/3314183.3324978>
- [29] Alejandro Infante and Alexandre Bergel. 2017. Object Equivalence: Revisiting Object Equality Profiling (An Experience Report). In *Proceedings of the 13th ACM SIGPLAN International Symp. on Dynamic Languages (DLS)*. 27–38. <https://doi.org/10.1145/3133841.3133844>
- [30] V. López Jaquero, F. Montero, J.P. Molina, and P. González. 2009. *Intelligent User Interfaces: Past, Present and Future*. Springer London, 1–12. https://doi.org/10.1007/978-1-84800-136-7_18
- [31] Monique W. M. Jaspers, Thiemo Steen, Cor van den Bos, and Maud M. Geenen. 2004. The Think Aloud Method: A Guide to User Interface Design. *I. J. Medical Informatics* 73, 11-12 (2004), 781–795. <https://doi.org/10.1016/j.ijmedinf.2004.08.003>
- [32] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated Performance Analysis of Load Tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*. 125–134. <https://doi.org/10.1109/ICSM.2009.5306331>
- [33] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why Don't Software Developers Use Atatic Analysis Tools to Find Bugs?. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*. 672–681. <https://doi.org/10.1109/ICSE.2013.6606613>
- [34] Vivien Johnston. 2019. A Framework for the Development of a Dynamic Adaptive Intelligent User Interface to Enhance the User Experience. In *Proceedings of the 31st European Conference on Cognitive Ergonomics (ECCE)*. 32–35. <https://doi.org/10.1145/3335082.3335125>
- [35] Reiner Jung and Marc Adolf. 2018. The JPetStore Suite: A Concise Experiment Setup for Research. In *Proceedings of the 9th Symposium on Software Performance (SSP)*. <http://eprints.uni-kiel.de/48775/>
- [36] Reiner Jung, Marc Adolf, and Christoph Dornieden. 2017. Towards Extracting Realistic User Behavior Models. *Softwaretechnik-Trends* 37, 3 (2017). <http://eprints.uni-kiel.de/40365/>
- [37] Marius Koller and Gerrit Meixner. 2016. Task Models in Practice: Are There Special Requirements for the Use in Daily Work?. In *Proceedings of 18th International Conference on Human-Computer Interaction (HCI) - Theory, Design, Development and Practice*. 488–497. https://doi.org/10.1007/978-3-319-39510-4_45
- [38] Steinar Kristoffersen. 2008. Learnability and Robustness of User Interfaces. Towards a Formal Analysis of Usability Design Principles. In *Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT), Volume SE/MUSE/GSDCA*. 261–268.
- [39] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*. 4:1–4:11. <https://doi.org/10.1145/2972206.2972220>
- [40] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 51–62. <https://doi.org/10.1145/2668930.2688037>
- [41] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 249–260. <https://doi.org/10.1145/2851553.2851555>
- [42] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 3–14. <https://doi.org/10.1145/3151759.3151795>

- [//doi.org/10.1145/3030207.3030211](https://doi.org/10.1145/3030207.3030211)
- [43] Jinbo Li, Hesam Izakian, Witold Pedrycz, and Iqbal Jamal. 2021. Clustering-based Anomaly Detection in Multivariate Time Series Data. *Appl. Soft Comput.* 100 (2021), 106919. <https://doi.org/10.1016/j.asoc.2020.106919>
 - [44] Jie Liang and Mao Lin Huang. 2010. Highlighting in Information Visualization: A Survey. In *Proceedings of the 14th International Conference on Information Visualisation (IV)*. 79–85. <https://doi.org/10.1109/IV.2010.21>
 - [45] Quentin Limbourg and Jean Vanderdonckt. 2003. *The Handbook of Task Analysis for Human-Computer Interaction*. CRC Press, Chapter Comparing Task Models for User Interface Design, 135–154.
 - [46] Víctor López-Jaquero and Francisco Montero Simarro. 2007. Comprehensive Task and Dialog Modelling. In *Proceedings of the 12th International Conference on Human-Computer Interaction (HCI) - Interaction Design and Usability*, Vol. 4550. Springer, 1149–1158. https://doi.org/10.1007/978-3-540-73105-4_125
 - [47] Darko Marinov and Robert O’Callahan. 2003. Object Equality Profiling. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. 313–325. <https://doi.org/10.1145/949305.949333>
 - [48] Evan K. Maxwell, Godmar Back, and Naren Ramakrishnan. 2010. Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 115–124. <https://doi.org/10.1145/1835804.1835822>
 - [49] Mark T. Maybury. 1999. Intelligent User Interfaces: An Introduction. In *Proceedings of the 4th International Conference on Intelligent User Interfaces (IUI)*. 3–4. <https://doi.org/10.1145/291080.291081>
 - [50] Karen L. McGraw and Bruce A. McGraw. 1997. Wizards, Coaches, Advisors, and More: A Performance Support Primer. In *Ext. Abstr. on Human Factors in Computing Systems (Atlanta, Georgia)*. 152–153. <https://doi.org/10.1145/1120212.1120318>
 - [51] Gerrit Meixner, Marc Seissler, and Kai Breiner. 2011. Model-Driven Useware Engineering. In *Model-Driven Development of Advanced User Interfaces*. 1–26. https://doi.org/10.1007/978-3-642-14562-9_1
 - [52] MyBatis. 2016. *JPetStore*. <http://mybatis.org/jpetstore-6/>
 - [53] Raymond H Myers and Raymond H Myers. 1990. *Classical and modern regression with applications*. Vol. 2. Duxbury press Belmont, CA.
 - [54] Jakob Nielsen. 1993. *Usability Engineering*. Academic Press.
 - [55] Mie Nørgaard and Kasper Hornbæk. 2006. What do Usability Evaluators do in Practice?: An Explorative Study of Think-aloud Testing. In *Proceedings of the Conference on Designing Interactive Systems (DIS)*. 209–218. <https://doi.org/10.1145/1142405.1142439>
 - [56] Oracle. 2020. *VisualVM: All-in-One Java Troubleshooting Tool*. <https://visualvm.github.io/>
 - [57] J. D. Ornelas, J. C. Silva, and J. L. Silva. 2016. USS: User support system. In *Proceedings of the 11th Iberian Conference on Information Systems and Technologies (CISTI)*. 1–6. <https://doi.org/10.1109/CISTI.2016.7521412>
 - [58] Fabio Paternò, Cristiano Mancini, and Silvia Meniconi. 1997. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*. 362–369.
 - [59] Wim De Pauw and Gary Sevitsky. 1999. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*. 116–134. https://doi.org/10.1007/3-540-48743-3_6
 - [60] Manjula Peiris and James H. Hill. 2016. Automatically Detecting "Excessive Dynamic Memory Allocations" Software Performance Anti-Pattern. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 237–248. <https://doi.org/10.1145/2851553.2851563>
 - [61] Aleksandar Prokopec, Andrea Rosà, David Leopoldseeder, Gilles Duboscq, Petr Tuma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 31–47. <https://doi.org/10.1145/3314221.3314637>
 - [62] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. 2017. A Comparison Framework for Runtime Monitoring Approaches. *J. Syst. Softw.* 125 (2017), 309–321. <https://doi.org/10.1016/j.jss.2016.12.034>
 - [63] Rick Rabiser, Klaus Schmid, Holger Eichelberger, Michael Vierhauser, Sam Guinea, and Paul Grünbacher. 2019. A Domain Analysis of Resource and Requirements Monitoring: Towards a Comprehensive Model of the Software Monitoring Domain. *Inf. Softw. Technol.* 111 (2019), 86–109. <https://doi.org/10.1016/j.infsof.2019.03.013>
 - [64] Mukund Raghothaman, Sulekha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 722–735. <https://doi.org/10.1145/3192366.3192417>
 - [65] Shaina Raza and Chen Ding. 2019. Progress in Context-aware Recommender Systems - An Overview. *Comput. Sci. Rev.* 31 (2019), 84–97. <https://doi.org/10.1016/j.cosrev.2019.01.001>

- [66] Steven P. Reiss. 2009. Visualizing the Java Heap to Detect Memory Problems. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. 73–80. <https://doi.org/10.1109/VISSOFT.2009.5336418>
- [67] J. Renz, T. Staubitz, J. Pollak, and C. Meinel. 2014. Improving the Onboarding User Experience in MOOCs. In *Proceedings of the 6th International Conference on Education and New Learning Technologies (EDULEARN)* (Barcelona, Spain). 3931–3941.
- [68] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2011. Elephant Tracks: Generating Program Traces with Object Death Records. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*. 139–142. <https://doi.org/10.1145/2093157.2093178>
- [69] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant Tracks: Portable Production of Complete and Precise GC Traces. In *Proceedings of the International Symposium on Memory Management (ISMM)*. 109–118. <https://doi.org/10.1145/2491894.2466484>
- [70] Cynthia K. Riemenschneider and Bill C. Hardgrave. 2001. Explaining Software Development Tool Use with the Technology Acceptance Model. *Journal of Computer Information Systems (JCIS)* 41, 4 (2001), 1–8. <https://www.tandfonline.com/doi/abs/10.1080/08874417.2001.11647015>
- [71] Roger C Schank, Tamara R Berman, and Kimberli A Macpherson. 1999. Learning by Doing. *Instructional-design theories and models: A new paradigm of instructional theory 2*, 2 (1999), 161–181.
- [72] Andreas Schörgenhumer, Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2017. Efficient Sampling-based Lock Contention Profiling for Java. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*. 331–334. <https://doi.org/10.1145/3030207.3030234>
- [73] Andreas Schörgenhumer, Mario Kahlhofer, Paul Grünbacher, and Hanspeter Mössenböck. 2019. Can we Predict Performance Events with Time Series Data from Monitoring Multiple Systems?. In *Companion of the ACM/SPEC International Conference on Performance Engineering ICPE*. 9–12. <https://doi.org/10.1145/3302541.3313101>
- [74] Andreas Schörgenhumer. 2017. *Efficient Sampling-based Lock Contention Profiling in Java*. Master’s thesis. Johannes Kepler University, Institute for System Software. <https://pub.jku.at/obvulihs/content/titleinfo/1825350>
- [75] Connie U. Smith and Lloyd G. Williams. 2000. Software Performance Antipatterns. In *Proceedings of the Second International Workshop on Software and Performance (WOSP)*. 127–136. <https://doi.org/10.1145/350391.350420>
- [76] Ken Soong, Xin Fu, and Yang Zhou. 2018. Optimizing New User Experience in Online Services. In *Proceedings of the 5th IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 442–449. <https://doi.org/10.1109/DSAA.2018.00057>
- [77] Miroslaw Staron, Wilhelm Meding, Jörgen Hansson, Christoffer Höglund, Kent Niesel, and Vilhelm Bergmann. 2014. Dashboards for Continuous Monitoring of Quality for Software Product under Development. In *Relating System Quality and Software Architecture*. 209–229. <https://doi.org/10.1016/b978-0-12-417009-4.00008-9>
- [78] Piyawadee Noi Sukaviriya and James D. Foley. 1990. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In *Proceedings of the 3rd Annual ACM Symp. on User Interface Software and Technology (UIST)*. 152–166. <https://doi.org/10.1145/97924.97942>
- [79] Claudia Szabo. 2015. Novice Code Understanding Strategies during a Software Maintenance Assignment. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*. 276–284. <https://doi.org/10.1109/ICSE.2015.341>
- [80] Eclipse Foundation. 2020. *Eclipse Memory Analyzer (MAT)*. <https://www.eclipse.org/mat/>
- [81] Doug Tidwell and Jeanette Fuccella. 1997. TaskGuides: Instant Wizards on the Web. In *Proceedings of the 15th Annual International Conference on Computer Documentation (SIGDOC)*. 263–272. <https://doi.org/10.1145/263367.263401>
- [82] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 762–774. <https://doi.org/10.1145/2660267.2660339>
- [83] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*. 213–222. <https://doi.org/10.1109/ICSM.2011.6080788>
- [84] Qingsong Wen, Jingkun Gao, Xiaomin Song, Liang Sun, Huan Xu, and Shenghuo Zhu. 2019. RobustSTL: A Robust Seasonal-Trend Decomposition Algorithm for Long Time Series. In *Proceedings of the Thirty-Third Conference on Artificial Intelligence (AAAI)*. 5409–5416. <https://doi.org/10.1609/aaai.v33i01.33015409>
- [85] Markus Weninger et al. 2020. *AntTracks*. <http://mevss.jku.at/AntTracks>
- [86] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proceedings of the 9th Symp. on Software Performance (SSP)*. 64–66. http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf
- [87] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proceedings of the 15th International*

- Conference on Managed Languages & Runtimes (ManLang)*. 14:1–14:13. <https://doi.org/10.1145/3237009.3237023>
- [88] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 273–284. <https://doi.org/10.1145/3297663.3310297>
- [89] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of Suspicious Time Windows In Memory Monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*. 95–104. <https://doi.org/10.1145/3357390.3361025>
- [90] Markus Weninger, Elias Ganer, and Hanspeter Mössenböck. 2020. Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance. In *Proceedings of the 11th Symp. on Software Performance (SSP)*. http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [91] Markus Weninger, Paul Grünbacher, Elias Gander, and Andreas Schörgenhumer. 2020. Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 75 (June 2020), 37 pages. <https://doi.org/10.1145/3394977>
- [92] Markus Weninger, Paul Grünbacher, Huihui Zhang, Tao Yue, and Shaukat Ali. 2018. Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. 21–30. <https://doi.org/10.1109/APSEC.2018.00016>
- [93] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. 357–360. <https://doi.org/10.1145/3030207.3030236>
- [94] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Comp. of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 29–32. <https://doi.org/10.1145/3302541.3313100>
- [95] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2019. Memory Leak Visualization using Evolving Software Cities. In *Proceedings of the 10th Symp. on Software Performance (SSP)*. 44–46. http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [96] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proceedings of the Working Conference on Software Visualization, (VISSOFT)*. 110–121. <https://doi.org/10.1109/VISSOFT51673.2020.00017>
- [97] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Heap Evolution Analysis Using Tree Visualizations. In *Proceedings of the 11th Symp. on Software Performance (SSP)*. http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf
- [98] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2020. Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations. In *Proceedings of the Conference on Smart Tools and Apps for Graphics - Eurographics Italian Chapter Conference*. <https://doi.org/10.2312/stag.20201241>
- [99] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*. 115–126. <https://doi.org/10.1145/3184407.3184412>
- [100] Jen-Her Wu and Yufei Yuan. 2003. Improving Searching and Reading Performance: The Effect of Highlighting and Text Color Coding. *Inf. Manag.* 40, 7 (2003), 617–637. [https://doi.org/10.1016/S0378-7206\(02\)00091-5](https://doi.org/10.1016/S0378-7206(02)00091-5)
- [101] Guoqing (Harry) Xu. 2013. Resurrector: a Tunable Object Lifetime Profiling Technique for Optimizing Real-World Programs. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 111–130. <https://doi.org/10.1145/2509136.2509512>
- [102] N. Zhang, N. Jiang, Y. Zhang, and G. Huang. 2010. Towards Automated Generation of User-Specific Eclipse Wizard. In *Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. 490–497. <https://doi.org/10.1109/CyberC.2010.95>

Received February 2021; accepted April 2021

Chapter 7

Memory Churn

This chapter includes the paper [320] that discusses advanced memory churn analysis using object lifetime information reconstructed from memory traces.

Paper:

Markus Weninger, Elias Gander, Hanspeter Mössenböck:

Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance. In *Proceedings of the 11th Symposium on Software Performance*, SSP 2020, Leipzig, Germany, November 12 - 13, 2020 (moved online).

Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance

Markus Weninger^{*}, Elias Gander[⊗], Hanspeter Mössenböck^{*}
{`firstname.lastname@jku.at`}

^{*} Institute for System Software, Johannes Kepler University, Linz, Austria

[⊗] Christian Doppler Laboratory MEVSS, Johannes Kepler University, Linz, Austria

Abstract

High memory churn occurs when many temporary objects are created and shortly thereafter collected by the garbage collector. Such excessive dynamic allocations negatively impact an application’s performance because (1) a great number of objects has to be allocated on the heap and (2) an increased number of garbage collections is required to collect them.

In this paper, we present ongoing research on how to support developers in detecting, understanding and resolving high memory churn in order to improve their application’s performance. Based on a recorded memory trace, an algorithm automatically searches for memory churn hotspots and calculates the age at which objects die within it, since objects that die young are the major contributors to memory churn. Information about these objects, for example their types and allocation sites, can then be inspected in order to locate the problematic code locations.

To demonstrate the feasibility and applicability of our approach, we implemented and present a new *memory churn analysis* feature in AntTracks, our trace-based memory monitoring tool.

1 Introduction

A common performance anti-pattern [1] is high memory churn. High memory churn, also known as excessive dynamic allocations [6], denotes the frequent creation and collection of objects. The time it takes to allocate these objects, as well as the time spent on collecting them during garbage collection, both negatively impact an application’s performance. Even though temporary objects often turn out to be superfluous and avoidable through minor adjustments of the underlying algorithms, most state-of-the-art memory monitoring tools do not provide analysis features to inspect memory churn in greater detail but rather focus on the analysis of memory leaks.

In this paper, we describe an approach to support developers during the investigation of high memory churn in garbage-collected languages. To motivate our work, we present typical causes for high memory churn in Section 2. In Section 3, we discuss how our approach automatically detects *memory churn hotspots*, i.e., time windows in which unusual amounts

of garbage is collected, based on the evolution of an application’s memory footprint. If a memory churn hotspot is detected, we calculate the *lifetime* of each object that died within the hotspot, as explained in Section 4. We perform this calculation since objects that die shortly after their creation are the main contributors to high memory churn. In Section 5, we discuss how lifetime information can be combined with information on other heap object properties (such as type or allocation site) to point users to code locations that should be inspected to reduce memory churn.

All concepts presented in this work have been implemented using our memory monitoring tool AntTracks¹. AntTracks encompasses two parts: (1) a modified Java VM that collects memory traces containing information about memory events such as allocations or garbage collections [5], and (2) an offline analysis tool that can reconstruct the monitored application’s heap states, i.e., the contents of the heap at different points in time, based on such a trace [7].

2 Motivation

The careless allocation of objects can lead to high memory churn that results in run-time overhead that could easily be prevented. A typical situation leading to high memory churn is the allocation of short-living temporary objects within heavily executed loops. Every iteration allocates new objects that quickly turn into garbage. Another typical problem is the use of boxed primitives as generic types, e.g., `ArrayList<Integer>`. Every time a primitive is added to such a structure, it is wrapped into a heap object, which causes unnecessary memory overhead. One last example is the careless use of streams. Often, multiple `map` operations (or similar) are used unnecessarily, causing many short-living intermediate objects to be created. Another classic mistake is to use `map` when working with primitives instead of using the respective memory-efficient operation such as `mapToInt`.

3 Memory Churn Hotspot Detection

The first step when checking an application for high memory churn is to look for *memory churn hotspots*. Figure 1 shows an application that exhibits frequent

¹AntTracks available at: <http://mevss.jku.at/AntTracks>

tall spikes in its memory footprint, a typical memory churn pattern. The plot depicts the monitored application’s memory footprint at the beginning and at the end of every garbage collection. Since the memory occupied at the start of a garbage collection is much higher than at its end, each garbage collection appears as the falling edge of a spike.

In [9], we presented algorithms to automatically detect suspicious patterns in an application’s memory footprint that hint at memory anomalies such as high memory churn. This feature aims to help novice users that would otherwise struggle to recognize problematic patterns on their own. Currently, we only present the most critical anomalies, e.g., the strongest memory churn hotspot, to not overwhelm the (novice) users with too much information. In general, the following steps are performed to find an application’s strongest memory churn hotspot (as done in Figure 1):

- Construct all possible time windows that cover between 5 and 50 garbage collections.
- Calculate each window’s *garbage per second* by summing the bytes collected within the window and dividing them by the window’s duration. Graphically speaking, sum the heights of all falling edges within the window and divide them by the window’s width.
- Finally, select the window with the highest garbage per second. If its garbage per second is significantly higher than the application’s average garbage per second, i.e., if it is a hotspot, report it (e.g., by highlighting in the plot).

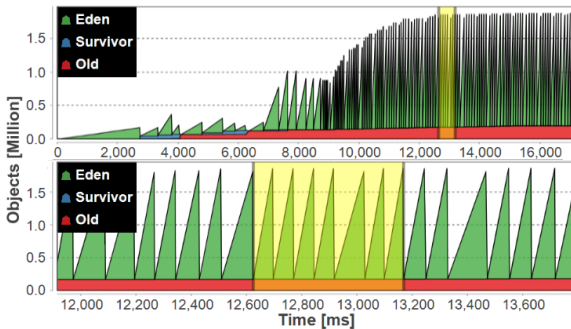


Figure 1: Automatically detected time window with high memory churn (global view and zoomed-in view).

4 Object Lifetime Calculation

To detect which objects die young, we need to know the time at which a given object was *born* and when it *died*. The *Mertlin* algorithm [2] used by *Elephant Tracks* [3] could be used to calculate very exact object death times, yet it causes a several 100-fold increase in the analyzed application’s run time [4]. Instead, we use less exact object ages, namely the *number of garbage collections an object survived*. This way, it is

sufficient to know for each object (1) the first garbage collection following its allocation and (2) during which garbage collection it was collected. Like most memory tracers, AntTracks records events at the start and the end of garbage collections, where garbage collections are assigned consecutive IDs. As shown in Figure 2, the *birth time* of each object is set to the ID of the garbage collection following the allocation. When the garbage collector reclaims an object, it is assigned the ID of the currently running garbage collection as its *free time*. It is then straightforward to calculate the age of a died object by subtracting the two IDs.

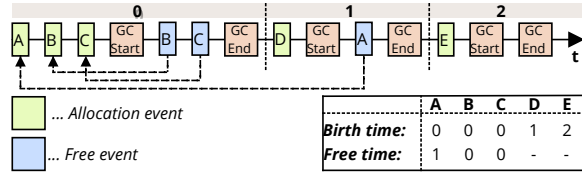


Figure 2: For every heap object, a *Birth Time* and *Free Time* is reconstructed.

5 Memory Churn Suspect Inspection

Many memory analysis tools, including AntTracks, group heap objects based on one or more criteria (such as their types) and display the number of objects and the number of bytes per heap object group [7]. This typically happens during heap state analysis, i.e., during the inspection of the heap at a given point in time.

We suggest a similar approach for memory churn analysis. Yet, instead of grouping the *live objects* at a given point in time, we group all objects that *died within a given time window*. This time window can be manually selected by the user or automatically detected, as explained in Section 3. We use a new grouping criterion, the *object lifetime grouping*, in addition to existing ones such as type and allocation site. As shown in Figure 3, this criterion aggregates the died objects into groups named “ $\langle x \rangle$ GCs survived”. Next to each group we display the number of objects and the number of bytes that have been collected by the garbage collector within the selected time window. Major memory churn contributors can be revealed by drilling down into the largest object groups that did not survive a single garbage collection.

6 Example

Figure 3 through Figure 5 show a complete example of how AntTracks has been used to investigate a memory churn hotspot in order to improve the *finagle-http* benchmark in the Renaissance benchmark suite [8] version 0.9.0. As shown in Figure 3, inspecting the automatically detected memory churn hotspot reveals that over 99.9% of the died objects (10,012,077 out of 10,019,784) did not even survive a single garbage collection. Inspecting the types of these objects in Figure 4 reveals that most of them are divided almost equally among four types (since *finagle-http* is a

Scala application, its type names are typically longer than Java type names). Next, we inspected the allocation sites of these four types and found out that the allocation sites of the first three types are within library methods which we cannot modify. Yet, Figure 5 shows that the allocation site of the fourth type (which are anonymous function objects) is located in the `FinagleHttp` class, the benchmark’s main class. Since such a rapid allocation and collection of anonymous function objects is unlikely to be intentional, we inspected the method’s source code. In a loop, a lot of anonymous function objects were created waiting for an HTTP request to succeed before incrementing a counter. In our fixed version, only a single response handler is created which is reused for every HTTP request. This reduced the overall amount of allocated temporary objects by about 25% and sped up the application by about 5%.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
4 GCs survived	7,686
1 GCs survived	21

Figure 3: Grouping objects by the number of survived GCs facilitates high memory churn analysis.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
Promise\$WaitQueue\$\$anon\$4	2,494,576
Promise\$Monitored	2,494,393
Future\$\$anonfun\$onSuccess\$1	2,494,362
FinagleHttp\$\$anonfun\$runIteration\$1\$\$anon\$2	2,494,361
char[]	3,196

Figure 4: Inspecting the types of the frequently dying objects reveals major suspect types.

Name	Collected objects
Overall	10,019,784
0 GCs survived	10,012,077
...	
FinagleHttp\$\$anonfun\$runIteration\$1\$\$anon\$2	2,494,361
FinagleHttp\$\$anonfun\$runIteration\$1\$\$anon\$2	2,494,361

Figure 5: The allocation sites of frequently dying objects lead to methods that have to be inspected.

7 Conclusion and Future Work

As high memory churn can have a substantial negative impact on an application’s performance, tool support to inspect such memory anomalies is essential. In this work, we discussed common causes for memory churn, we showed how to automatically detect memory churn hotspots, we presented how to detect objects that die shortly after their allocation, and suggested a way how to utilize and visualize this information for memory churn analysis. To showcase the applicability of our approach, we implemented it in our memory monitoring tool AntTracks and presented an example on how the tool’s new memory churn analysis feature has been used to improve a real-world benchmark application.

For future work, we currently focus on making AntTracks (including its new memory churn analysis) more accessible to novice users. As we evaluated AntTracks’s various capabilities [10], we observed a need for more guidance during memory anomaly analysis tasks. This led us to elaborate recommendations for memory monitoring tool developers including ‘*Use automation to relieve users from complex tasks*’ as well as ‘*Provide guidance and explanations to support exploratory learning of analysis capabilities*’.

8 Acknowledgement

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and Dynatrace is gratefully acknowledged.

References

- [1] C. U. Smith and L. G. Williams. “Software Performance Antipatterns”. In: *WOSP*. 2000, pp. 127–136.
- [2] M. Hertz et al. “Generating Object Lifetime Traces with Merlin”. In: *ACM Trans. Program. Lang. Syst.* 28.3 (May 2006), pp. 476–516.
- [3] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. “Elephant Tracks: Portable Production of Complete and Precise GC Traces”. In: *ISMM*. 2013, pp. 109–118.
- [4] G. Xu. “Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs”. In: *OOPSLA*. 2013, pp. 111–130.
- [5] P. Lengauer, V. Bitto, and H. Mössenböck. “Accurate and Efficient Object Tracing for Java Applications”. In: *ICPE*. 2015, pp. 51–62.
- [6] M. Peiris and J. H. Hill. “Automatically Detecting ”Excessive Dynamic Memory Allocations” Software Performance Anti-Pattern”. In: *ICPE*. 2016, pp. 237–248.
- [7] M. Weninger and H. Mössenböck. “User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring”. In: *ICPE*. 2018, pp. 115–126.
- [8] A. Prokopec et al. “Renaissance: Benchmarking Suite for Parallel Applications on the JVM”. In: *PLDI*. 2019, pp. 31–47.
- [9] M. Weninger, E. Gander, and H. Mössenböck. “Detection of Suspicious Time Windows In Memory Monitoring”. In: *MPLR*. 2019, pp. 95–104.
- [10] M. Weninger et al. “Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study”. In: *Proc. ACM Hum.-Comput. Interact.* 4.EICS (June 2020), 75:1–75:37.

Part III

Future Work and Conclusions

Chapter 8

Limitations and Future Work

Although we presented various ways how to analyze memory anomalies in garbage-collected languages, there is still a number of interesting topics that can and should be explored in the future. In this section, we describe some of them and how future work could remove current limitations of this thesis.

8.1 Memory Anomaly Evaluation Suite

All papers in Part II contained evaluations to showcase the approaches' feasibility and applicability, most of them based on either artificial programs [78] or performance benchmark applications [25, 230, 265, 293, 295]. Only few evaluations happened based on real-world applications. In this section, we discuss why it is hard to find suitable real-world applications to evaluate memory anomaly analyses and how we hope to resolve this issue.

While artificial programs are well-suited to explain concepts, they often do not show if an approach is generalizable and scalable. Artificial programs do not show whether the presented approach would also work well on real-world applications or if the artificially generated problems even occur in real-world applications.

Thus, performance benchmark suites such as DaCapo [25], DaCapo Scala [265, 293], SPECjvm2008 [295] or Renaissance [230] try to include benchmark applications that mimic real-world behavior. Yet, while these suites have been used to study general memory and garbage collection behavior [176], they are not well-suited to evaluate *anomaly* analysis

techniques due to the simple fact that these benchmark applications hardly contain any memory defects.

Various software repository mining studies [65, 100, 101] suggest that there are a number of real-world, open-source applications that exhibited at least one memory-related problem throughout their development. Unfortunately, there is no ready-to-use memory anomaly evaluation suite that contains such reference applications that could serve as a basis for evaluating and comparing memory anomaly analyses. Currently, one has to put a lot of work into locating and preparing a single application so that it can be used as evaluation reference, for example as we did with the memory-leaking *Commons HttpClient* library [7, 8, 324]. First, one has to mine software repositories and issue trackers to find notes or issues on memory problems that have been resolved in the past. Then, the source code of the respective application has to be downloaded in the version that still contained the memory defect before it was fixed. Next, the project has to be built, which, depending on the project size, can involve quite complicated dependencies and build steps. Overall, finding and preparing suitable applications for evaluation is time-consuming and tedious.

Thus, to build a common ground that can be used to evaluate and compare memory anomaly analyses, it would be desirable to collect a suite of memory anomaly evaluation applications that contain known memory defects. We think that the memory research community could greatly profit from such a suite. It could be used to re-evaluate the presented approaches in greater detail, to more thoroughly compare these approaches to others, and to perform further user studies that do not have to rely on seeded defects.

8.2 Automatic Data Structure Detection

In our current data structure analysis approach, we rely on a domain-specific language to describe how data structures look like. Even though this approach works well and is easy to use, it would be more convenient if this step was not needed, i.e., if AntTracks could detect data structures on its own without external input.

While related work exists that tries to do exactly this (for example various work by Mitchell et al. [197, 198, 200–202] or by Chis et al. [56, 57]), all of them have their own shortcomings or drawbacks, which we will discuss shortly.

Existing approaches either use the object reference graph directly or reduce it to some kind of ownership tree and try to detect data structures based on certain patterns within these graphs or trees. For example, typical heuristics to detect data structure heads are to look for an object that either (a) points to an array of reference or (b) points to an object of type T which in turn points to another object of type T. This would detect data structures that use an array or linked objects as a backbone. Similar pattern definitions are then given for *backbone* structures and *leaf* structures, where the number and naming of these structures differs from approach to approach. Yet, even though many data structures may fit these heuristics, it seems that no universal approach has been found yet that is able to detect all kinds of data structures. For example, one simple data structure that many (if not all) of the cited approaches cannot detect automatically is Java’s `HashSet`. In Java, hash sets are implemented by an underlying hash map, i.e., the hash set object (which should be classified as a *head*) directly points to another data structure head. Data structure heads directly pointing to other data structure heads are a pattern that most algorithms do not handle correctly. For example, Mitchell and Sevitsky state that they *use framework knowledge to classify the `HashSet` as “head of collection”, though it would not normally be classified as such solely on the basis of structure* [201].

Thus, future research could use the mentioned related work as an inspiration for more advanced, automatic data structure detection algorithms. We suggest to use the correct detection of Java’s default data structures as well as the correct detection of the most-widely used third-party data structures in Java [65] (without manual intervention such as for `HashSet`) as a baseline.

8.3 Lifetime Analysis

In our work on memory churn analysis, we presented the concept of *heap object lifetime*, i.e., how many garbage collections an object survived before it died. This metric may not only be useful for memory churn analysis, but also for various other kinds of analyses, which could be explored in future research.

One idea is to refine our data structure analysis by lifetime information. Currently, we detect growing data structures, report them to the user and show *quantitative information*, e.g., how many leaves of a certain type have been there at the beginning and at the end of a given time window. Taking

lifetime information into account, we could inform the user also about those leaf objects that got *older*, since old objects that have been part of a growing data structure for a long time are more likely to be part of a possible memory leak.

8.4 Metric-based Analysis

An interesting topic that we did not explore in greater detail yet revolves around general memory anti-patterns [56, 57, 135]. For example, Chis et al. [57] present 11 patterns such as *empty collections* or *fixed-size collections* (instead of arrays) that should be avoided. In their work, they use heap dumps to analyze applications regarding these metrics, i.e., they calculate these metrics at a single point in time. While these metrics seem to be a great way to evaluate the memory health of an application, the usage of heap dumps leads to the problem that a certain pattern may be present coincidentally at one point in time, even though this would not be the case moments later. For example, taking a heap snapshot at the wrong point in time may detect a lot of empty collections, yet they might be false positives since they might be filled with objects a few moments later.

Our idea is to perform similar metric-based analyses, however not only at a single point in time but continuously based on data reconstructed from memory traces. The more often and the more strongly a certain data structure exhibits a given inefficiency pattern, the more confidently we could report it as worth for inspection. This would eliminate the unnecessary task from the user to distinguish critical from non-critical pattern occurrences.

8.5 Visualization Extensions

In this thesis, various visualization techniques such as time-series charts, memory cities, 2D memory tree visualizations and aggregated object graphs have been shown. Future work could improve and extend these visualizations into various directions. For example, even though our memory city approach supports the visualization of references, its focus is on depicting the evolution of the heap memory over time. For reference visualization, hierarchical edge bundling [47, 123, 124, 126] or pipe routing [21] has been employed in various domains in the past. We could investigate if these techniques or similar

approaches can also be used to visualize heap memory and its references. In addition to that, there are also some other visualization techniques that could be evaluated whether they are suitable for visualizing heap memory evolution visualization, one of them being small multiple visualizations [302] to display the evolution of different parts of the heap side-by-side. Also, the presented visualization techniques could be integrated more strongly into our data structure analysis, using them to visualize the growth of data structures in a more tangible way. The usefulness of such visualizations could then be evaluated in further user studies [298].

8.6 Using Visualizations in SE Education

We think that our visualizations may not only be helpful to developers to inspect memory anomalies in their applications, but also to students who learn about garbage collection behavior or performance engineering. From our experience, such topics are often taught in a rather theoretical way, using few to no visualizations of real applications. This may also be due to the fact that state-of-the-art monitoring tools (if used for demonstration at all) often lack interactive visualization techniques. We think that visualizations such as our memory cities could be used to explain the rationale behind certain memory pattern in an interesting and memorable way. Memory anomalies that are usually only explained in a theoretical context become tangible, i.e., they are raised from a purely syntactical form to a graphical metaphor. Studying and understanding memory behavior by using visualizations in software engineering education could lead to higher student motivation, better learning outcomes and hopefully higher student success rates.

8.7 Static and Dynamic Analysis Synergies

Currently, our approach is purely based on information reconstructed from a memory trace recorded during an application run, i.e., it can be classified as a pure *dynamic analysis*. In contrast, as explained in Section 1.3, static analysis does not rely on data collected during program execution but rather inspects the source code (or other static artifacts). In the past, both kinds of analyses have been combined [76, 81] to achieve synergies by merging data from static and dynamic analyses.

One technique where we think that such a *hybrid* or *blended* analysis could lead to more detailed insights is our memory churn analysis. Based on our experience, we are confident to say that memory churn hotspots are often also reflected in the source code. For example, many memory churn hotspots are the result of heavily-executed loops that contain allocations of short-living objects. As we have shown, our approach is able to detect in which method problematic objects are created, yet suggestions on how to actually *resolve* the problem would require us to have knowledge about the actual source code, i.e., information that we could acquire through static analysis. Knowing whether the allocation site of churning heap objects is located within a loop or in the context of some other typical memory churn pattern would allow us to provide the developer with more detailed instructions on how to resolve the problem.

8.8 IDE Integration

To end this section on future work, we want to present our hypothesis that users could greatly profit from monitoring tools that were integrated more tightly with IDEs. Taking AntTracks as an example, there is a clear separation between (a) the (guided) analyses within the monitoring tool that may lead the user to the root cause of a problem, and (b) the (unguided) inspection and fixing steps the developers have to perform in the IDE to resolve the given problem. For example, even after developers used AntTracks to narrow the possible root cause of a problem down to a single allocation site, they still have to fix the source code in their IDE without further guidance. In the future, research should be performed on how information gathered through monitoring approaches can be efficiently and effectively fed back into IDEs (scientific contribution), where developing an IDE plugin [16, 60] for AntTracks (technical contribution) could serve as a case study.

Chapter 9

Conclusions

Even though garbage collection reduces the risk of memory-related defects, certain performance-critical anomalies such as high memory churn or even crash-prone anomalies such as memory leaks can still occur. To help developers in detecting, understanding and fixing these anomalies, this thesis has contributed in various areas of memory analysis using memory traces.

Memory Traces and Their Processing First, we have presented a novel and versatile grouping system to abstract a heap state into a *memory tree* based on a set of *object classifiers* that group heap objects with respect to certain properties. This grouping technique allows us to pursue a user-centered analysis approach in which the users can decide according to which heap object properties they want to inspect a heap state. During inspection, the users can drill down into the tree's different object groups to gather more information about them. Furthermore, throughout this thesis we have shown that our memory trees are a versatile data basis for a vast amount of different (semi-)automatic analyses and visualizations.

Memory traces can not only be used for extracting information to build memory trees, but they also provide information about the references between heap objects and how these references change over time. To utilize this information to its full potential, we further AntTracks's existing tracing technique to also collect information about garbage collection (GC) roots such as thread-local variables or static fields. This GC root information, together with information about the object references, allows us to pinpoint objects that keep a large number of other objects alive. We also presented

algorithms that focus on *multi-object ownership* analysis, a problem that is widely overlooked in related work.

Data Structure Analysis Second, we presented an approach that automatically detects heavily-growing data structures, ranks them based on various growth metrics, and displays them to the user for more detailed investigation. For this, we developed a domain-specific language, which can be used to describe the shape of arbitrary data structures. Using these descriptions, our approach detects data structures in heap states and — due to continuous data that can be reconstructed from memory traces — is able to track these structures as well as their growth across their lifetime. Depending on the kind of growth (which can be categorized based on metrics we developed in this thesis) users are suggested further problem-specific analysis steps to find out more about the root cause of the growth.

Visualization As many studies and related work have shown, interactive visualizations can greatly enhance analysis tasks. However, most state-of-the-art memory monitoring tools lack good visualization support. Thus, as a third topic, we extensively investigated how memory trees and their evolution over time can be visualized to support developers in detecting and analyzing suspicious memory behavior. To this end, we developed three novel techniques to visualize evolving memory trees: (1) Time-series charts with a drill-down function, (2) *Memory Cities*, a 3D software city visualization in which growing buildings and districts hint at growing object groups in the heap, as well as (3) traditional 2D tree visualizations, namely icicles and sunbursts, that have been adjusted and refined to better suit memory analysis tasks. Our memory cities and tree visualizations were both honored with a best paper award at international conferences, which gives us confidence that the presented ideas can be useful for memory analysis, even though detailed studies have yet to be performed.

User Guidance and User Behavior While many memory analysis approaches have been evaluated with regard to their performance, i.e., how long the analyses take to produce a result, there are no proper studies on the usability of memory analysis tools. We filled this gap by performing a cognitive walkthrough as well as a user study where AntTracks was used to investigate typical memory anomalies. As the study subjects were no memory analysis

experts, this study provided us with valuable insights on how novices use memory tools and how to support them in a better way. This also led us to the formulation of nine recommendations for monitoring tool developers when implementing new analysis features. These recommendations already influenced a number of improvements to AntTracks itself, most importantly a guidance method called *guided exploration*. Guided exploration is a method for integrating user guidance into monitoring tools by repeatedly performing four guidance steps: Tools should automatically (1) *detect* suspicious memory behavior, (2) *highlight* information about it on the screen, (3) *explain* why this information is important, and (4) *suggest* appropriate next analysis steps. This way, the tool *guides* users through the analysis process, helping them to *explore* the root cause of a problem. At the same time, users learn the capabilities of the tool and how to use them efficiently through a learning-by-doing effect. This should ease the onboarding process for novice users in expert monitoring tools.

Memory Churn Analysis Combining our multi-level heap object grouping mechanism with our guided exploration helped us to develop a convenient analysis technique to detect, inspect and resolve memory churn, i.e., the frequent allocation and deallocation of short-living objects. Our approach automatically detects time windows in which the monitored application exhibited suspicious churn behavior, detects those objects that died at a young age within this time window, and guides the user to those object types and allocation sites that caused the churn. AntTracks also provides information and explanations on how memory churn can be resolved and how typical memory churn hotspots look like. These features can help novice users to fix memory churn anomalies, even without a prior background in memory monitoring.

Summarized Contributions Overall, this thesis presented (1) algorithms and data structures to aggregate and utilize memory traces, (2) a semi-automatic approach to detect and inspect suspiciously growing data structures, (3) visualization techniques to make memory evolution and memory growth analysis more approachable and tangible, (4) a technique to detect and inspect high memory churn, (5) a cognitive walkthrough and a user study to evaluate the usability and usefulness of the presented techniques, as well as to derive general recommendations for developers of memory monitoring

tools, which finally led to (6) a guidance technique called *guided exploration* that supports novice users in performing memory analyses.

One goal of this work was to highlight the flexibility and versatility of memory traces. We wanted to present interesting use cases on how they can be leveraged to support developers in finding, inspecting and fixing memory anomalies. Thus, we have shown that temporal information, i.e., information about the heap evolution over time, can provide detailed insights into memory trends. Continuous memory information is vital to perform detailed memory analyses such as automatic data structure analysis. Such analyses would not be possible by only relying on heap dumps, since dumps lack object identity information. Even though the generation of memory traces is currently not widespread in real-world virtual machines, we hope that research such as this thesis can serve as a motivation to change that.

In addition to theoretical concepts, this thesis resulted in a number of technical contributions, namely the improvement of the *AntTracks VM*, the development of the *AntTracks Analyzer* tool, our 3D visualization tool *Memory Cities*, and our web-based 2D visualization tool *WebTreeViz*, which are all publicly available. We used these tools as a proof-of-concept for all presented ideas to underline their feasibility, applicability and usefulness.

Appendices

Appendix A

Memory Cities Artifact

This instruction document has been submitted as part of our Memory Cities artifact [329] that accompanied our paper *Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor* [326]. The artifact contains the executable binaries (Windows and Linux), data sets, a video, and the instruction document.

Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor

Artifact Instructions

Markus Weninger
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
markus.weninger@jku.at

Lukas Makor
CD Laboratory MEVSS
Johannes Kepler University Linz
Linz, Austria
lukas.makor@jku.at

Hanspeter Mössenböck
Institute for System Software
Johannes Kepler University Linz
Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract—This document contains instruction on how to use the artifact accompanying the paper “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor” presented at VISSOFT 2020. The following abstract is taken from the paper.

Tool support is essential to help developers understand the memory behavior of complex software systems. Anomalies such as memory leaks can dramatically impact application performance and can even lead to crashes. Unfortunately, most memory analysis tools lack advanced visualizations (especially of the memory evolution over time) that could facilitate developers in analyzing suspicious memory behavior.

In this paper, we present *Memory Cities*, a technique to visualize an application’s *heap memory evolution* over time using the *software city* metaphor. While this metaphor is typically used to visualize static artifacts of a software system such as class hierarchies, we use it to visualize the dynamic memory behavior of an application. In our approach, heap objects can be grouped by multiple properties such as their types or their allocation sites. The resulting object groups are visualized as buildings arranged in districts, where the size of a building corresponds to the number of heap objects or bytes it represents. Continuously updating the city over time creates the immersive feeling of an evolving city. This can be used to detect and analyze memory leaks, i.e., to search for suspicious growth behavior. Memory cities further utilize various visual attributes to ease this task. For example, they highlight strongly growing buildings using color, while making less suspicious buildings semi-transparent.

We implemented memory cities as a standalone application developed in Unity, with a JSON-based interface to ensure easy data import from external tools. We show how memory cities can use data provided by AntTracks, a trace-based memory monitoring tool, and present case studies on different applications to demonstrate the tool’s applicability and feasibility.

Index Terms—Artifact

I. PREFACE

We prepared this artifact according to <http://vissoft20.dcc.uchile.cl/submission.html>. The submission guidelines ask for artifacts that primarily contain *software systems* (compiled applications and/or source code) that are available for end users and researchers who aim at replicating a study, as well as *repositories* that contain data involved in a study (e.g., logging data, system traces, survey raw data, evaluation results), which are needed to replicate a study.

As *software system*, we provide a Windows build as well as a Linux build of the Memory Cities tool. The tool has been developed using the 3D engine Unity [1]. To follow the artifact guidelines at <https://www.artifact-eval.org/guidelines.html>, we planned to create a VM (Virtualbox/VMWare) image containing the Memory Cities tool already setup in the intended runtime environment. Unfortunately, the 3D acceleration needed to run Memory Cities does not integrate well with virtual machines. An alternative WebGL build of Memory Cities was not feasible since the tool needs access to the local file system, a feature not well supported by WebGL.

As *repository*, we provide the data used to create Figure 10 (Motivating Example), Figure 11 (Case Study 1) and Figure 12 (Case Study 2) in the original paper. Section V lists the settings used within Memory Cities to reproduce these figures based on the provided data.

II. CLAIM

The builds provided in this artifact are slightly polished versions of those that have been used to create the screenshots of the original paper. Users can thus expect the same visuals as presented in the paper. All features discussed in the paper (e.g., navigation, interaction, heap reference analysis, etc.) are fully implemented. Only two settings, namely the evolution animation speed and the switch between the object and byte metric, have been mentioned in the paper to be user-defined but are currently only configurable in code. The tool is still under development, and both settings will be available in the next release that contains a refactored settings pane.

Regarding reproducibility, we provide usage instructions in Section IV as well as detailed steps to reproduce Figure 10, Figure 11 and Figure 12 of the original paper in Section V.

We want to mention that, while Memory Cities are fully functional and ready to use, we have not yet focused on the tool’s performance. We are aware that certain parts in the tool (e.g., the outline drawing of buildings) have the potential for performance improvements in the future.

The tool has been primarily tested on Windows, but also all test runs on Linux (Ubuntu) were successful.



Fig. 1. When the tool is started, it asks the user to select the folder that contains the memory tree data that should be visualized. In the screenshot, clicking “Select” twice loads the data and builds the memory city to reconstruct Figure 10 of the original paper.

III. ARTIFACT CONTENT

This artifact is provided as a .zip file that contains the following files and directories:

- 1) Instructions.pdf, this file.
- 2) Paper.pdf, the original paper [2]. This documents assumes that the reader is familiar with the content of the paper.
- 3) MemoryCities-Video.mp4, a 5 minute long video that explains the basics of the Memory Cities tool¹. It is recommended to watch the video before using the tool.
- 4) MemoryCities-Windows, a folder that contains the Windows build, most importantly the executable MemoryCities.exe to run the tool on Windows.
- 5) MemoryCities-Linux, a folder that contains the Linux build, most importantly the script Run.sh to run the tool on a *nix system. If any permission problems occur, please use chmod to mark the files in this directory as executable, for example by running `chmod -R 777 .` within the directory.
- 6) PaperData, a folder that contains the subfolders Fig10, Fig11, and Fig12, which contain the data used to generate the respective figures in the original paper.

¹The video can also be found online at <http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.mp4>

IV. INSTRUCTION

A. How To Load Data

Upon opening, the tool asks the user to load data based on which a memory city should be generated. Figure 1 shows the respective folder chooser. In the provided dataset, the following folders contain the data used to reconstruct the respective figures of the original paper:

- PaperData -> Fig10 -> PackageType
- PaperData -> Fig11 -> TypeAllocationSite
- PaperData -> Fig12 -> TypeClosestDomainCallSite

We added a sub-folder, e.g., TypeAllocationSite, for every figure to clarify based on which properties the heap objects have been grouped - in the case of TypeAllocationSite by types (districts) and allocation sites (buildings). These folders contain a number of .json files that contain information on the grouped heap at different points in time. The files’ format corresponds to the one explained in Section IV-B of the original paper.

In Memory Cities, the data is split into (1) *tree data* that is mandatory to create a city, and (2) optional *pointer information* (as discussed in section VII-D of the original paper). Thus, a second folder chooser is shown that allows the user to select a folder that contains a `pointed-from-maps` and a

Mouse interaction:
 Mouse wheel - Forward / Backward, Drag pressed mouse wheel - Up / Down / Left / Right,
 Drag right mouse button - Rotate, Click left mouse button - Select / Deselect building
 Keyboard interaction:
 W - Forward, A - Left, S - Backward, D - Right, Q - Up, E - Down,
 B - Bird's eye view, R - Reset, F - Focus selected building
 P - Previous time, N - Next time

Fig. 2. The navigation hints shown in Memory Cities.

points-to-maps folder. For the provided dataset, the same folder as in the previous step can be selected. For example, to reconstruct Figure 10 of the original paper, both folder choosers can be maneuvered to PaperData -> Fig10 -> PackageType as shown in Figure 1. If the user does not want to load pointer information, the second file chooser can be closed by clicking on “Cancel”.

B. Evolution Visualization: Time Travel

To visualize the memory evolution over time, Memory Cities apply time traveling, i.e., stepping back and forth through the memory history of a system while the city updates itself to reflect the current state. As shown in Figure 3, time stepping can be performed manually using buttons or a slider as well as using the arrow keys on the keyboard. Additionally, the evolution can also be animated automatically. Users can pause and restart the animation at any point in time.



Fig. 3. Time travel controls to step through time.

C. Navigation

As listed in Figure 2 (a screenshot taken from the tool), the camera can be rotated using the right mouse button and zoomed using the mouse wheel. By dragging the mouse while the mouse wheel is pressed or by using the keyboard, the user can move the camera. Memory cities also provide keyboard shortcuts for typical tasks. For example, pressing the B key moves the camera into a bird's eye view, which can be useful to inspect the district structure.

D. Structure Information

Hovering over a building or district displays information about its respective heap object group. This information includes the path from the tree root, e.g., Heap -> Type: Person -> Allocation Site: foo(), the number of objects and the number of bytes, as shown in Figure 4.

Besides showing a structure's information on hover, users can also click on a structure to highlight it, which is also shown in Figure 4. The structure stays selected when moving through time to make it easier to track its evolution.

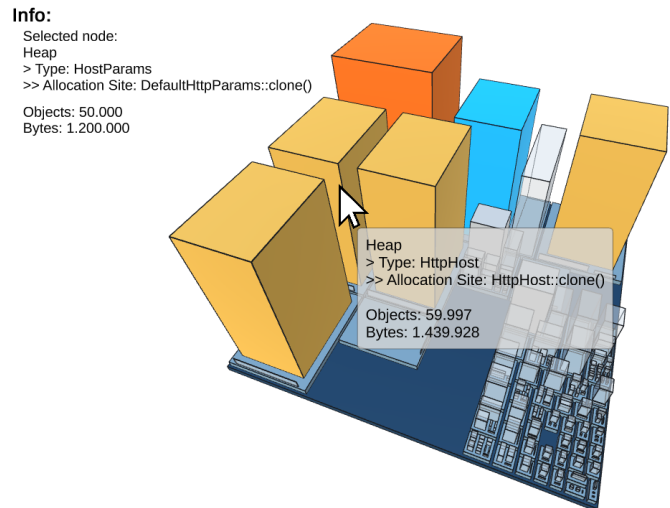


Fig. 4. Information about a structure is shown when hovering it (gray tool tip) or when selecting it with a click (selected building is highlighted in blue).

E. Settings

Figure 5 shows the settings available on the right side of the Memory Cities tool to adjust the appearance of the city.

a) *Area mode*: Determines the relation between object count and building size. The “Linear” setting directly maps the object count to building size (i.e., an increase of objects by a factor of 2 results in a building with a base area twice as big). The “Sqrt” setting calculates a building's size based on the square root of the object count of the underlying heap object group and often leads to more appealing-looking cities.

b) *Child count*: Determines how many children are shown per tree level, e.g., how many buildings are shown per district.

c) *Show pointers*: Enables / Disables the heap object reference feature that is explained in detail in the original paper in section VII-D. In general, if this feature is enabled, clicking on a building will show references between buildings. Buildings that contain objects that reference objects in the selected building are connected via a purple frustum. Buildings that contain objects that are referenced by objects in the select

building are connected via a green frustum. Thick frustums represent that more objects are involved in the references.

d) *Color (Growth)*: Enables / Disables the coloring of buildings. The color encodes how strong a building has grown between the first point in time and the current point in time.

e) *Solid buildings*: The last two settings, i.e., *Solid building count* and *Non-solid building opacity*, determine how many buildings should be drawn in solid mode, and which opacity non-solid buildings should have. Buildings are selected for solid mode based on their overall growth, i.e., the growth between the first and the last point in time. For example, setting *Solid building count* to 5 and *Non-solid building opacity* to 0 will draw all except the 5 strongest growing buildings transparent, while setting *Non-solid building opacity* to 100 will cause all buildings to be drawn in solid mode.

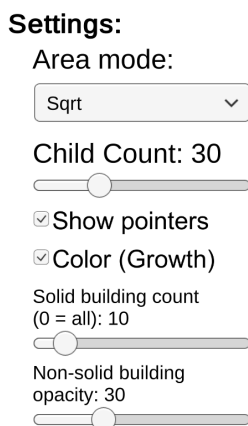


Fig. 5. The available settings to adjust the appearance of Memory Cities.

V. REPRODUCE FIGURES

This section lists all settings to reconstruct Figure 10, Figure 11, and Figure 12 of the original paper. Minimal differences between the figures in the original paper and the cities reconstructed from the data in this artifact can occur. This is due to the fact that the data provided in this artifact, i.e., information about the memory evolution of a monitored application, has been recorded on the different application run than the data used to generate the paper figures. Since the garbage collector behaves slightly differently on two different runs of the same application (e.g., it is triggered at different times), the resulting cities can vary a little from run to run.

After data loading, specific buildings can be located by hovering over them and inspecting their information popup. Clicking on a building with the left mouse button shows the building's references.

Figure 10

Title: LinkedList of Persons.

Classifiers: Package (Districts) - Type (Buildings)

Figure 10 (included in this document as Figure 6) is used in the original paper as a motivating example for the heap reference analysis.

To reconstruct this figure, load the data (in both folder choosers) from PaperData -> Fig10 -> PackageType. Following settings have been used:

- Time: 23/23
- Area Mode: Sqrt
- Child Count: 10
- Show Pointers: True
- Color: True
- Solid Buildings: 10
- Non-solid building opacity: 0%

Figure 11

Title: Commons HttpClient memory leak.

Classifiers: Type (Districts) - Allocation Site (Buildings)

Figure 11 (included in this document as Figure 7) is the first case study presented in the original paper. It demonstrates the analysis of a memory leak in the Commons HttpClient library.

To reconstruct this figure, load the data (in both folder choosers) from PaperData -> Fig11 -> TypeAllocationSite. Following settings have been used during heap reference analysis:

- Time: 19/25
- Area Mode: Sqrt
- Child Count: 40
- Show Pointers: True
- Color: True
- Solid Buildings: 10
- Non-solid building opacity: 30%

Figure 12

Title: Dynatrace easyTravel memory leak.

Classifiers: Type (Districts) - Closest Domain Call Site (Buildings)

Figure 12 (included in this document as Figure 8) is the second case study presented in the original paper. It demonstrates the analysis of a memory leak in the Dynatrace easyTravel application.

To reconstruct this figure, load the data (in both folder choosers) from PaperData -> Fig12 -> TypeClosestDomainCallSite. Following settings have been used during heap reference analysis:

- Time: 24/24
- Area Mode: Sqrt
- Child Count: 30
- Show Pointers: True
- Color: True
- Solid Buildings: 10
- Non-solid building opacity: 25%

REFERENCES

- [1] Unity Technologies. (2020) Unity Website. [Online]. Available: <https://unity.com>
- [2] M. Weninger, L. Makor, and H. Mössenböck, "Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor," in *Proc. of the 8th IEEE Working Conference on Software Visualization (VISSOFT)*, 2020.

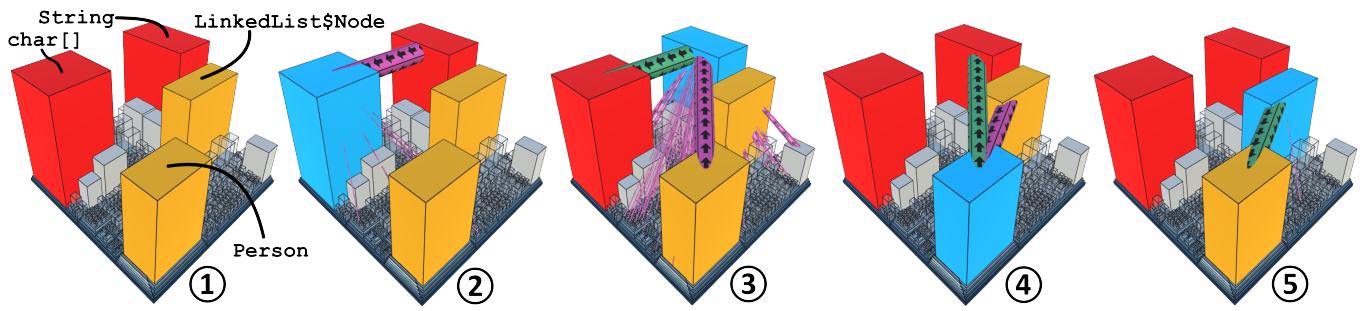


Fig. 6. Figure 10 of the original paper [2].

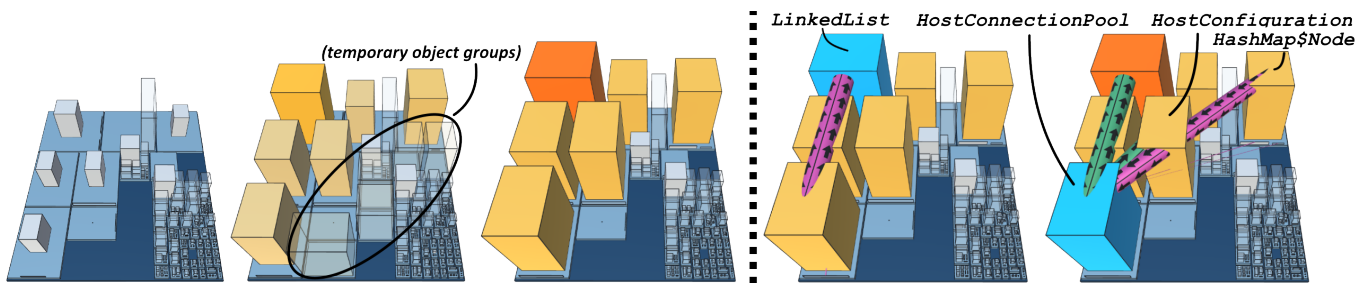


Fig. 7. Figure 11 of the original paper [2].

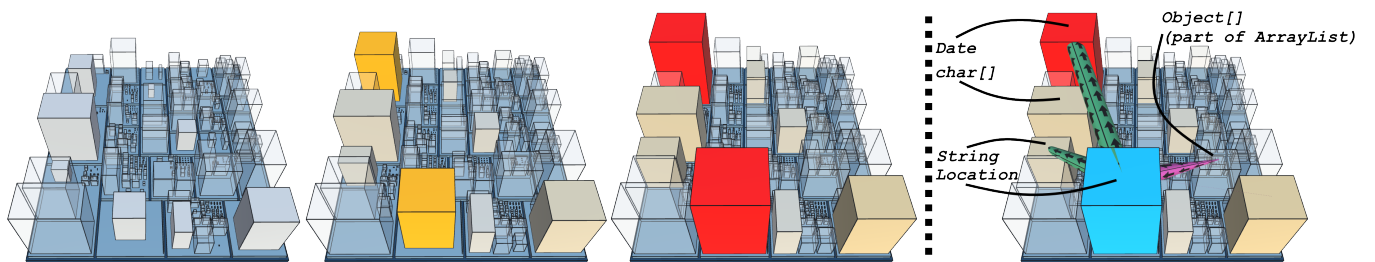


Fig. 8. Figure 12 of the original paper [2].

Bibliography

- [1] A. Abran, A. Khelifi, W. Suryn, and A. Seffah. Usability Meanings and Interpretations in ISO Standards. *Software Quality Journal*, 11(4):325–338, 2003. DOI: <https://www.doi.org/10.1023/A:1025869312943>.
- [2] E. Aftandilian, S. Kelley, C. Gramazio, N. P. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. In *Proceedings of the ACM Symposium on Software Visualization (SOFTVIS)*, pages 53–62, 2010. DOI: <https://www.doi.org/10.1145/1879211.1879222>.
- [3] T. M. Ahmed, C. Bezemer, T. Chen, A. E. Hassan, and W. Shang. Studying the Effectiveness of Application Performance Management (APM) Tools for Detecting Performance Regressions for Web Applications: An Experience Report. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR)*, pages 1–12, 2016. DOI: <https://www.doi.org/10.1145/2901739.2901774>.
- [4] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A Query Language for Manipulating Object-oriented Databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases (VLDB)*, pages 433–442, 1989. URL: <http://www.vldb.org/conf/1989/P433.PDF>.
- [5] S. Alstrup and P. W. Lauridsen. A Simple Dynamic Algorithm for Maintaining a Dominator Tree. Technical report, 1996.
- [6] K. Andrews and H. Heidegger. Information Slices: Visualising and Exploring Large Hierarchies using Cascading, Semi-Circular Discs. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 9–12, 1998.
- [7] Apache Software Foundation. Commons HttpClient version 3.0.1, 2006. URL: <https://mvnrepository.com/artifact/commons-httpclient/commons-httpclient/3.0.1>. last visited on 2021-04-08.
- [8] Apache Software Foundation. Issue tracker for HttpClient, 2021. URL: <https://issues.apache.org/jira/projects/HTTPCLIENT/issues>. last visited on 2021-04-08.
- [9] D. Auber, C. Huet, A. Lambert, B. Renoust, A. Sallaberry, and A. Saulnier. GosperMap: Using a Gosper Curve for Laying Out Hierarchical Data. *IEEE Trans. Vis. Comput. Graph.*, 19(11):1820–1832, 2013. DOI: <https://www.doi.org/10.1109/TVCG.2013.91>.
- [10] Audacity. Audacity: Free, open source, cross-platform audio software, 2021. URL: <https://www.audacityteam.org/>. last visited on 2021-04-08.
- [11] I. Bacher, B. M. Namee, and J. D. Kelleher. Using Icicle Trees to Encode the Hierarchical Structure of Source Code. In *EuroVis*, pages 97–101, 2016. DOI: <https://www.doi.org/10.2312/eurovisshort.20161168>.
- [12] G. Balogh and Á. Beszédés. CodeMetropolis - Code Visualisation in Minecraft. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 136–141, 2013. DOI: <https://www.doi.org/10.1109/SCAM.2013.6648194>.
- [13] G. Balogh and Á. Beszédés. CodeMetropolis - A Minecraft based Collaboration Tool for Developers. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013. DOI: <https://www.doi.org/10.1109/VISSOFT.2013.6650528>.
- [14] G. Balogh, T. Gergely, Á. Beszédés, and T. Gyimóthy. Using the City Metaphor for Visualizing Test-Related Metrics. In *Proceedings of the International Workshop on Validating Software Tests (VST@SANER)*, pages 17–20, 2016. DOI: <https://www.doi.org/10.1109/SANER.2016.48>.
- [15] G. Balogh, A. Szabolcs, and Á. Beszédés. CodeMetropolis: Eclipse over the City of Source Code. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 271–276, 2015. DOI: <https://www.doi.org/10.1109/SCAM.2015.7335425>.

- [16] S. Baltes, P. Schmitz, and S. Diehl. Linking Sketches and Diagrams to Source Code Artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 743–746, 2014. DOI: <https://www.doi.org/10.1145/2635868.2661672>.
- [17] S. T. Barlow and P. Neville. A Comparison of 2-D Visualizations of Hierarchies. In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS)*, pages 131–138, 2001. DOI: <https://www.doi.org/10.1109/INFVIS.2001.963290>.
- [18] E. T. Barr, C. Bird, and M. Marron. Collecting a Heap of Shapes. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133. ACM, 2013. DOI: <https://www.doi.org/10.1145/2483760.2483761>.
- [19] A. Bauer, M. Züfle, J. Grohmann, N. Schmitt, N. Herbst, and S. Kounev. An Automated Forecasting Framework based on Method Recommendation for Seasonal Time Series. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 48–55. ACM, 2020. DOI: <https://www.doi.org/10.1145/3358960.3379123>.
- [20] M. Bellingham, S. Holland, and P. Mulholland. A Cognitive Dimensions Analysis of Interaction Design for Algorithmic Composition Software. In *Proceedings of the 25th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, page 18, 2014.
- [21] G. Belov, W. Du, M. G. de la Banda, D. Harabor, S. Koenig, and X. Wei. From Multi-Agent Pathfinding to 3D Pipe Routing. In *Proceedings of the International Symposium on Combinatorial Search (SOCS)*, pages 11–19, 2020. URL: <http://aaai.org/ocs/index.php/SOCS/SOCS20/paper/view/18513>.
- [22] V. Bitto and P. Lengauer. Building Custom, Efficient, and Accurate Memory Monitoring Tools for Java Applications. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 321–324, 2016. DOI: <https://www.doi.org/10.1145/2851553.2858664>.
- [23] V. Bitto, P. Lengauer, and H. Mössenböck. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proceedings of the International Conference on Principles and Practices of Programming on The Java Platform (PPPJ)*, pages 76–89, 2015. DOI: <https://www.doi.org/10.1145/2807426.2807433>.
- [24] R. P. Biuk-Aghai, P. C. Pang, and B. Pang. Map-like Visualisations vs. Treemaps: An Experimental Comparison. In *Proceedings of the 10th International Symposium on Visual Information Communication and Interaction (VINCI)*, pages 113–120, 2017. DOI: <https://www.doi.org/10.1145/3105971.3105976>.
- [25] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006. DOI: <https://www.doi.org/10.1145/1167473.1167488>.
- [26] A. Blackwell and T. Green. CHAPTER 5 - Notational Systems—The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*, Interactive Technologies, pages 103 – 133. 2003. DOI: <https://www.doi.org/10.1016/B978-155860808-5/50005-8>.
- [27] A. F. Blackwell. Cognitive Dimensions of Notations. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, page 3, 2005. DOI: <https://www.doi.org/10.1109/VLHCC.2005.26>.
- [28] A. F. Blackwell. Cognitive Dimensions of Notations: Understanding the Ergonomics of Diagram Use. In *Proceedings of the 5th International Conference on Diagrammatic Representation and Inference*, pages 5–8, 2008. DOI: https://www.doi.org/10.1007/978-3-540-87730-1_4.
- [29] A. F. Blackwell, C. Britton, A. L. Cox, T. R. G. Green, C. A. Gurr, G. F. Kadoda, M. Kutar, M. Loomes, C. L. Nehaniv, M. Petre, C. Roast, C. Roe, A. Wong, and R. M. Young. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Proceedings of the 4th Conference on Cognitive Technology*, pages 325–341, 2001. DOI: https://www.doi.org/10.1007/3-540-44617-6_31.
- [30] T. Bladh, D. A. Carr, and J. Scholl. Extending Tree-Maps to Three Dimensions: A Comparative Study. In *Proceedings of the 6th Asia Pacific Conference on Computer Human Interaction (APCHI)*, pages 50–59, 2004. DOI: https://www.doi.org/10.1007/978-3-540-27795-8_6.
- [31] A. F. Blanco, J. P. S. Alcocer, and A. Bergel. Effective Visualization of Object Allocation Sites. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, pages 43–53, 2018. DOI: <https://www.doi.org/10.1109/VISSOFT.2018.00013>.
- [32] J. Bohnet and J. Döllner. Monitoring Code Quality and Development Activity by Software Maps. In *Proceedings of the 2nd Workshop on Managing Technical Debt (MTD)*, pages 9–16, 2011. DOI: <https://www.doi.org/10.1145/1985362.1985365>.

- [33] M. D. Bond and K. S. McKinley. Bell: Bit-encoding Online Memory Leak Detection. In J. P. Shen and M. Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 61–72, 2006. DOI: <https://www.doi.org/10.1145/1168857.1168866>.
- [34] M. D. Bond and K. S. McKinley. Tolerating Memory Leaks. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 109–126, 2008. DOI: <https://www.doi.org/10.1145/1449764.1449774>.
- [35] M. D. Bond and K. S. McKinley. Leak Pruning. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–288, 2009. DOI: <https://www.doi.org/10.1145/1508244.1508277>.
- [36] G. Booch, J. E. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide - Covers UML 2.0 (Second Edition)*. Addison-Wesley, 2005.
- [37] M. Bostock. d3.js, 2021. URL: <https://d3js.org/>. last visited on 2021-04-08.
- [38] M. Bostock, V. Ogievetsky, and J. Heer. D³ Data-Driven Documents. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2301–2309, 2011. DOI: <https://www.doi.org/10.1109/TVCG.2011.185>.
- [39] J. Boyle and P. M. D. Gray. The Design of 3D Metaphors for Database Visualisation. In *Proceedings of the 3rd IFIP 2.6 Working Conference on Visual Database Systems*, volume 34, pages 185–202, 1995. DOI: https://www.doi.org/10.1007/978-0-387-34905-3_12.
- [40] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe for the Past: Adding Genericity to the Java Programming Language. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 183–200, 1998. DOI: <https://www.doi.org/10.1145/286936.286957>.
- [41] M. Bruls, K. Huizing, and J. J. van Wijk. Squarified Treemaps. In *Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization (VisSym)*, pages 33–42, 2000. DOI: https://www.doi.org/10.1007/978-3-7091-6783-0_4.
- [42] E. Bruneton, E. Kuleshov, A. Loskutov, and R. Forax. ASM, 2021. URL: <https://asm.ow2.io/>. last visited on 2021-04-08.
- [43] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. *Adaptable and Extensible Component Systems*, 30(19), 2002.
- [44] J. Brutlag. Speed Matters for Google Web Search, 2009. URL: <https://ai.googleblog.com/2009/06/speed-matters.html>. last visited on 2021-04-08.
- [45] J. Calleya, R. Pawling, C. Ryan, and H. M. Gaspar. Using Data Driven Documents (D3) to Explore a Whole Ship Model. In *Proceedings of the 11th System of Systems Engineering Conference (SoSE)*, pages 1–6, 2016. DOI: <https://www.doi.org/10.1109/SYSOSE.2016.7542947>.
- [46] P. Caserta and O. Zendra. Visualization of the Static Aspects of Software: A Survey. *IEEE Trans. Vis. Comput. Graph.*, 17(7):913–933, 2011. DOI: <https://www.doi.org/10.1109/TVCG.2010.110>.
- [47] P. Caserta, O. Zendra, and D. Bodenès. 3D Hierarchical Edge bundles to Visualize Relations in a Software City Metaphor. In *Proceedings of the 6th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VIS-SOFT)*, pages 1–8, 2011. DOI: <https://www.doi.org/10.1109/VISSOF.2011.6069451>.
- [48] I. Cassar, A. Francalanza, L. Aceto, and A. Ingólfssdóttir. A Survey of Runtime Monitoring Instrumentation Techniques. In *Proceedings of the Second International Workshop on Pre- and Post-Deployment Verification Techniques (PrePost)*, volume 254, pages 15–28, 2017. DOI: <https://www.doi.org/10.4204/EPTCS.254.2>.
- [49] R. Cattell, D. K. Barry, M. Berler, J. Eastman, D. Jordan, C. Russell, O. Schadow, T. Stanienda, and F. Velez. *The Object Data Standard: ODMG 3.0*. 2000.
- [50] N. Cawthon and A. V. Moere. The Effect of Aesthetic on the Usability of Data Visualization. In *Proceedings of the 11th International Conference on Information Visualisation (IV)*, pages 637–648, 2007. DOI: <https://www.doi.org/10.1109/IV.2007.147>.
- [51] K. Chen and J. Chen. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, pages 23–28, 2007. DOI: <https://www.doi.org/10.1109/COMPSAC.2007.79>.
- [52] S. Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 313–336, 2000. DOI: https://www.doi.org/10.1007/3-540-45102-1_16.
- [53] S. Chiba. Javassist, 2020. URL: <https://www.javassist.org/>. last visited on 2021-04-08.

- [54] S. Chiba and M. Nishizawa. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE)*, pages 364–376, 2003. DOI: https://www.doi.org/10.1007/978-3-540-39815-8_22.
- [55] T. M. Chilimbi, R. E. Jones, and B. G. Zorn. Designing a Trace Format for Heap Allocation Events. In *Proc. of the International Symposium on Memory Management (ISMM)*, pages 35–49, 2000. DOI: <https://www.doi.org/10.1145/362422.362435>.
- [56] A. E. Chis. Automatic Detection of Memory Anti-patterns. In *Companion Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 925–926, 2008. DOI: <https://www.doi.org/10.1145/1449814.1449911>.
- [57] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O’Sullivan, T. Parsons, and J. Murphy. Patterns of Memory Inefficiency. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 6813, pages 383–407, 2011. DOI: https://www.doi.org/10.1007/978-3-642-22655-7_18.
- [58] M. Christakis and C. Bird. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 332–343, 2016. DOI: <https://www.doi.org/10.1145/2970276.2970347>.
- [59] M. C. Chuah. Dynamic Aggregation with Circular Visual Designs. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis)*, pages 35–43, 1998. DOI: <https://www.doi.org/10.1109/INFVIS.1998.729557>.
- [60] J. Cito, P. Leitner, C. Bosshard, M. Knecht, G. Mazlami, and H. C. Gall. PerformanceHat: Augmenting Source Code with Runtime Performance Traces in the IDE. In *Companion Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 41–44, 2018. DOI: <https://www.doi.org/10.1145/3183440.3183481>.
- [61] C. Click, G. Tene, and M. Wolf. The Pauseless GC Algorithm. In M. Hind and J. Vitek, editors, *Proceedings of the 1st International Conference on Virtual Execution Environments (VEE)*, pages 46–56, 2005. DOI: <https://www.doi.org/10.1145/1064979.1064988>.
- [62] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.
- [63] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution Trace Analysis Through Massive Sequence and Circular Bundle Views. *J. Syst. Softw.*, 81(12):2252–2268, 2008. DOI: <https://www.doi.org/10.1016/j.jss.2008.02.068>.
- [64] B. Cornelissen, A. Zaidman, A. van Deursen, and B. V. Rompaey. Trace Visualization for Program Comprehension: A Controlled Experiment. In *Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC)*, pages 100–109, 2009. DOI: <https://www.doi.org/10.1109/ICPC.2009.5090033>.
- [65] D. Costa, A. Andrzejak, J. Seboek, and D. Lo. Empirical Study of Usage and Performance of Java Collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 389–400, 2017. DOI: <https://www.doi.org/10.1145/3030207.3030221>.
- [66] D. Costa and R. Matias Jr. Characterization of Dynamic Memory Allocations in Real-World Applications: An Experimental Study. In *Proceedings of the 23rd IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 93–101, oct 2015. DOI: <https://www.doi.org/10.1109/MASCOTS.2015.28>.
- [67] K. Cox. Cognitive Dimensions of Use Cases: Feedback from a Student Questionnaire. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, page 8, 2000. URL: <http://ppig.org/library/paper/cognitive-dimensions-use-cases-feedback-student-questionnaire>.
- [68] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An Efficient Method of Computing Static Single Assignment Form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 25–35, 1989. DOI: <https://www.doi.org/10.1145/75277.75280>.
- [69] F. D. Davis. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly*, 13(3):319–340, 1989. URL: <http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html>.
- [70] W. De Pauw and G. Sevitsky. Visualizing Reference Patterns for Solving Memory Leaks in Java. *Concurrency - Practice and Experience*, 12(14):1431–1454, 2000. DOI: [https://www.doi.org/10.1002/1096-9128\(20001210\)12:14<1431::AID-CPE542>3.0.CO;2-2](https://www.doi.org/10.1002/1096-9128(20001210)12:14<1431::AID-CPE542>3.0.CO;2-2).

- [71] D. Detlefs, C. H. Flood, S. Heller, and T. Printezis. Garbage-first Garbage Collection. In D. F. Bacon and A. Diwan, editors, *Proceedings of the 4th International Symposium on Memory Management (ISMM)*, pages 37–48, 2004. DOI: <https://www.doi.org/10.1145/1029873.1029879>.
- [72] D. Distefano and I. Filipovic. Memory Leaks Detection in Java by Bi-abductive Inference. In D. S. Rosenblum and G. Taentzer, editors, *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 6013, pages 278–292, 2010. DOI: https://www.doi.org/10.1007/978-3-642-12029-9_20.
- [73] S. dos Santos and K. Brodlie. Gaining Understanding of Multivariate and Multidimensional Data Through Visualization. *Computers & Graphics*, 28(3):311 – 325, 2004. DOI: <https://www.doi.org/http://doi.org/10.1016/j.cag.2004.03.013>.
- [74] D. C. Dryer. Wizards, Guides, and beyond: Rational and Empirical Methods for Selecting Optimal Intelligent User Interface Agents. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces (IUI)*, pages 265–268, 1997. DOI: <https://www.doi.org/10.1145/238218.238347>.
- [75] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic Metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 149–168, 2003. DOI: <https://www.doi.org/10.1145/949305.949320>.
- [76] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended Analysis for Performance Understanding of Framework-based Applications. In D. S. Rosenblum and S. G. Elbaum, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 118–128, 2007. DOI: <https://www.doi.org/10.1145/1273463.1273480>.
- [77] Dynatrace. Dynatrace, 2021. URL: <https://www.dynatrace.com/>. last visited on 2021-04-08.
- [78] Dynatrace. Dynatrace Demo Application: easyTravel, 2021. URL: <https://github.com/Dynatrace/easyTravel-Docker>. last visited on 2021-04-08.
- [79] Eclipse Foundation. Eclipse Memory Analyzer (MAT), 2021. URL: <https://www.eclipse.org/mat/>. last visited on 2021-04-08.
- [80] ej-technologies. JProfiler, 2021. URL: <https://www.ej-technologies.com/products/jprofiler/overview.html>. last visited on 2021-04-08.
- [81] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Proceedings of the Workshop on Dynamic Analysis (WODA)*, pages 24–27, May 2003.
- [82] J. Eve and R. Kurki-Suonio. On Computing the Transitive Closure of a Relation. *Acta Informatica*, 8:303–314, 1977. DOI: <https://www.doi.org/10.1007/BF00271339>.
- [83] R. Falke, R. Klein, R. Koschke, and J. Quante. The Dominance Tree in Visualizing Software Dependencies. In *Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–6, 2005. DOI: <https://www.doi.org/10.1109/VISSOFT.2005.1684311>.
- [84] A. Felfernig, G. Ninaus, H. Grabner, F. Reinfrank, L. Weninger, D. Pagano, and W. Maalej. An Overview of Recommender Systems in Requirements Engineering. In *Managing Requirements Knowledge*, pages 315–332. 2013. DOI: https://www.doi.org/10.1007/978-3-642-34419-0_14.
- [85] I. Fette and A. Melnikov. The WebSocket Protocol. *RFC*, 6455:1–71, 2011. DOI: <https://www.doi.org/10.17487/RFC6455>.
- [86] F. Fittkau, S. Finke, W. Hasselbring, and J. Waller. Comparing Trace Visualizations for Program Comprehension Through Controlled Experiments. In *Proceedings of the 23rd IEEE International Conference on Program Comprehension (ICPC)*, pages 266–276, 2015. DOI: <https://www.doi.org/10.1109/ICPC.2015.37>.
- [87] F. Fittkau, A. Krause, and W. Hasselbring. Exploring Software Cities in Virtual Reality. In *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT)*, pages 130–134, 2015. DOI: <https://www.doi.org/10.1109/VISSOFT.2015.7332423>.
- [88] F. Fittkau, A. Krause, and W. Hasselbring. Hierarchical Software Landscape Visualization for System Comprehension: A Controlled Experiment. In *Proceedings of the 3rd IEEE Working Conference on Software Visualization (VISSOFT)*, pages 36–45, 2015. DOI: <https://www.doi.org/10.1109/VISSOFT.2015.7332413>.
- [89] F. Fittkau, A. Krause, and W. Hasselbring. Software Landscape and Application Visualization for System Comprehension with ExplorViz. *Inf. Softw. Technol.*, 87:259–277, 2017. DOI: <https://www.doi.org/10.1016/j.infsof.2016.07.004>.
- [90] F. Fittkau, S. Roth, and W. Hasselbring. ExplorViz: Visual Runtime Behavior Analysis of Enterprise Application Landscapes. In *Proceedings of the European Conference on Information Systems (ECIS)*, 2015. URL: http://aisel.aisnet.org/ecis2015_cr/46.

- [91] F. Fittkau, P. Stelzer, and W. Hasselbring. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *Proceedings of the European Conference on Software Architecture (ECSA)*, pages 28:1–28:4, 2014. DOI: <https://www.doi.org/10.1145/2642803.2642831>.
- [92] F. Fittkau, A. van Hoorn, and W. Hasselbring. Towards a Dependability Control Center for Large Software Landscapes. In *Proceedings of the 10th European Dependable Computing Conference (EDCC)*, pages 58–61, 2014. DOI: <https://www.doi.org/10.1109/EDCC.2014.12>.
- [93] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. In *Proceedings of the 1st IEEE Working Conference on Software Visualization (VIS-SOFT)*, pages 1–4, 2013. DOI: <https://www.doi.org/10.1109/VISSOFT.2013.6650536>.
- [94] C. H. Flood, R. Kennke, A. Dinn, A. Haley, and R. Westrelin. Shenandoah: An Open-source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*, pages 13:1–13:9, 2016. DOI: <https://www.doi.org/10.1145/2972206.2972210>.
- [95] E. Folmer and J. Bosch. Usability Patterns in Software Architecture. In *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI)*, pages 93–97, 2003. URL: <http://www.grise.upm.es/rearviewmirror/projects/status/results/patterns.pdf>.
- [96] T. Fu. A Review on Time Series Data Mining. *Eng. Appl. Artif. Intell.*, 24(1):164–181, 2011. DOI: <https://www.doi.org/10.1016/j.engappai.2010.09.007>.
- [97] E. Gamma, R. Helm, R. E. Johnson, and J. M. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP)*, pages 406–431, 1993. DOI: https://www.doi.org/10.1007/3-540-47910-4_21.
- [98] E. Gander. Extending the Memory Monitoring Tool AntTracks to Guide Users During Heap Evolution Analysis. Master’s thesis, Johannes Kepler University, Institute for System Software, 2020. URL: <https://epub.jku.at/obvulihs/content/titleinfo/4741346>.
- [99] J. G. García, J. Vanderdonckt, and C. Lemaigre. Identification Criteria in Task Modeling. In *Proceedings of the 1st TC 13 IFIP Human-Computer Interaction Symposium (HCIS)*, volume 272, pages 7–20, 2008. DOI: https://www.doi.org/10.1007/978-0-387-09678-0_2.
- [100] M. Ghanavati, D. Costa, A. Andrzejak, and J. Seboek. Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *Companion Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 410–411, 2018. DOI: <https://www.doi.org/10.1145/3183440.3195032>.
- [101] M. Ghanavati, D. Costa, J. Seboek, D. Lo, and A. Andrzejak. Memory and Resource Leak Defects and Their Repairs in Java Projects. *Empir. Softw. Eng.*, 25(1):678–718, 2020. DOI: <https://www.doi.org/10.1007/s10664-019-09731-8>.
- [102] D. Gilbert. JFreeChart, 2021. URL: <http://www.jfree.org/jfreechart/>. last visited on 2021-04-08.
- [103] D. Gilbert. JFreeChart-FX, 2021. URL: <https://github.com/jfree/jfreechart-fx>. last visited on 2021-04-08.
- [104] Google. Android Studio, 2021. URL: <https://developer.android.com/studio>. last visited on 2021-04-08.
- [105] B. Göschlberger and P. A. Bruck. Gamification in Mobile and Workplace Integrated Microlearning. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services (iiWAS)*, pages 545–552, 2017. DOI: <https://www.doi.org/10.1145/3151759.3151795>.
- [106] T. Green. Instructions and Descriptions: Some Cognitive Aspects of Programming and Similar Activities. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI)*, pages 21–28, 2000. DOI: <https://www.doi.org/10.1145/345513.345233>.
- [107] T. Green and A. Blackwell. Cognitive Dimensions of Information Artefacts: a tutorial, 1998. URL: <https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf>. last visited on 2021-04-08.
- [108] T. R. G. Green. Cognitive Dimensions of Notations. In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*, pages 443–460, 1989. URL: <http://dl.acm.org/citation.cfm?id=92968.93015>.
- [109] B. Gregg. The Flame Graph. *ACM Queue*, 14(2):10, 2016. DOI: <https://www.doi.org/10.1145/2927299.2927301>.
- [110] B. Gregg. The Flame Graph. *Commun. ACM*, 59(6):48–57, 2016. DOI: <https://www.doi.org/10.1145/2909476>.
- [111] T. Hagos. *Android Studio IDE Quick Reference: A Pocket Guide to Android Studio Development*. Apress, 2019.

- [112] S. Hahn, J. Trümper, D. Moritz, and J. Döllner. Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps. In *Proceedings of the 5th International Conference on Information Visualization Theory and Applications (IVAPP)*, pages 50–58, 2014. DOI: <https://www.doi.org/10.5220/0004686200500058>.
- [113] J. Hamari, J. Koivisto, and H. Sarsa. Does Gamification Work? - A Literature Review of Empirical Studies on Gamification. In *Proceedings of the 47th Hawaii International Conference on System Sciences (HICSS)*, pages 3025–3034, 2014. DOI: <https://www.doi.org/10.1109/HICSS.2014.377>.
- [114] M. Hauswirth and T. M. Chilimbi. Low-overhead Memory Leak Detection Using Adaptive Statistical Profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 156–164, 2004. DOI: <https://www.doi.org/10.1145/1024393.1024412>.
- [115] J. Heer, M. Bostock, and V. Ogievetsky. A Tour through the Visualization Zoo. *ACM Queue*, 8(5):20, 2010. DOI: <https://www.doi.org/10.1145/1794514.1805128>.
- [116] W. E. Hefley and D. Murray. Intelligent User Interfaces. In *Proceedings of the 1st International Conference on Intelligent User Interfaces (IUI)*, page 3–10, 1993. DOI: <https://www.doi.org/10.1145/169891.169892>.
- [117] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free Garbage Collection Traces: How to Cheat and Not Get Caught. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 140–151, 2002. DOI: <https://www.doi.org/10.1145/511334.511352>.
- [118] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating Object Lifetime Traces with Merlin. *ACM Trans. Program. Lang. Syst.*, 28(3):476–516, May 2006. DOI: <https://www.doi.org/10.1145/1133651.1133654>.
- [119] T. Hill, J. Noble, and J. Potter. Scalable Visualisations with Ownership Trees. In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS)*, pages 202–213, 2000. DOI: <https://www.doi.org/10.1109/TOOLS.2000.891370>.
- [120] T. Hill, J. Noble, and J. Potter. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.*, 13(3):319–339, 2002. DOI: <https://www.doi.org/10.1006/jvlc.2002.0238>.
- [121] A. Hiniker, S. R. Hong, Y. Kim, N. Chen, J. D. West, and C. R. Aragon. Toward the Operationalization of Visual Metaphor. *J. Assoc. Inf. Sci. Technol.*, 68(10):2338–2349, 2017. DOI: <https://www.doi.org/10.1002/asi.23857>.
- [122] P. Hofer, D. Gnedt, A. Schörgenhumer, and H. Mössenböck. Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 263–274, 2016. DOI: <https://www.doi.org/10.1145/2851553.2851559>.
- [123] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Trans. Vis. Comput. Graph.*, 12(5):741–748, 2006. DOI: <https://www.doi.org/10.1109/TVCG.2006.147>.
- [124] D. Holten, B. Cornelissen, and J. J. van Wijk. Trace Visualization Using Hierarchical Edge Bundles and Massive Sequence Views. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 47–54, 2007. DOI: <https://www.doi.org/10.1109/VISSOFT.2007.4290699>.
- [125] A. Holzinger. Usability Engineering Methods for Software Developers. *Commun. ACM*, 48(1):71–74, 2005. DOI: <https://www.doi.org/10.1145/1039539.1039541>.
- [126] W. Hop, S. de Ridder, F. Frasinca, and F. Hogenboom. Using Hierarchical Edge Bundles to Visualize Complex Ontologies in GLOW. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 304–311, 2012. DOI: <https://www.doi.org/10.1145/2245276.2245338>.
- [127] M. Höst, B. Regnell, and C. Wohlin. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering*, 5(3):201–214, Nov 2000.
- [128] M. Hucko, L. Gazo, P. Simún, M. Valky, R. Móro, J. Simko, and M. Bieliková. YesElf: Personalized Onboarding for Web Applications. In *Adjunct Publications of the 27th Conference on User Modeling, Adaptation and Personalization (UMAP)*, pages 39–44. ACM, 2019. DOI: <https://www.doi.org/10.1145/3314183.3324978>.
- [129] A. Infante and A. Bergel. Object Equivalence: Revisiting Object Equality Profiling (An Experience Report). In *Proceeding of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*, pages 27–38, 2017. DOI: <https://www.doi.org/10.1145/3133841.3133844>.
- [130] T. Janjusic and K. Kavi. Gleipnir: A Memory Profiling and Tracing Tool. *SIGARCH Comput. Archit. News*, 41(4):8–12, Dec. 2013. DOI: <https://www.doi.org/10.1145/2560488.2560491>.

- [131] V. L. Jaquero, F. Montero, J. Molina, and P. González. *Intelligent User Interfaces: Past, Present and Future*, pages 1–12. 2009. DOI: https://www.doi.org/10.1007/978-1-84800-136-7_18.
- [132] M. W. M. Jaspers, T. Steen, C. van den Bos, and M. M. Geenen. The think aloud method: A guide to user interface design. *I. J. Medical Informatics*, 73(11-12):781–795, 2004. DOI: <https://www.doi.org/10.1016/j.ijmedinf.2004.08.003>.
- [133] JavaMelody. JavaMelody : monitoring of JavaEE applications, 2020. URL: <https://github.com/javamelody/javamelody/wiki>. last visited on 2021-04-08.
- [134] S. Jayaraman, B. Jayaraman, and D. Lessa. Compact Visualization of Java Program Execution. *Softw. Pract. Exp.*, 47(2):163–191, 2017. DOI: <https://www.doi.org/10.1002/spe.2411>.
- [135] K. Jezek and R. Lipka. Antipatterns Causing Memory Bloat: A Case Study. In *Proceedings of the 24th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 306–315, 2017. DOI: <https://www.doi.org/10.1109/SANER.2017.7884631>.
- [136] Z. M. Jiang, A. E. Hassan, G. Hamann, and P. Flora. Automated Performance Analysis of Load Tests. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, pages 125–134, 2009. DOI: <https://www.doi.org/10.1109/ICSM.2009.5306331>.
- [137] Jikes RVM Team. The Jikes RVM (Research VM) Project, 2016. URL: <https://www.jikesrvm.org>. last visited on 2021-04-08.
- [138] B. Johnson and B. Shneiderman. Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures. In *Proceedings of the IEEE Conference on Visualization*, pages 284–291, 1991. DOI: <https://www.doi.org/10.1109/VISUAL.1991.175815>.
- [139] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge. Why Don’t Software Developers Use Atatic Analysis Tools to Find Bugs? In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 672–681, 2013. DOI: <https://www.doi.org/10.1109/ICSE.2013.6606613>.
- [140] V. Johnston. A Framework for the Development of a Dynamic Adaptive Intelligent User Interface to Enhance the User Experience. In *Proceedings of the 31st European Conference on Cognitive Ergonomics (ECCE)*, pages 32–35, 2019. DOI: <https://www.doi.org/10.1145/3335082.3335125>.
- [141] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC, 2016.
- [142] R. E. Jones and C. Ryder. A Study of Java Object Demographics. In *Proceedings of the 7th International Symposium on Memory Management (ISMM)*, pages 121–130, 2008. DOI: <https://www.doi.org/10.1145/1375634.1375652>.
- [143] L. J. W. Jr. Visual Metaphors for Teaching Programming Concepts. In *Proceedings of the 20th SIGCSE Technical Symposium on Computer Science Education (SIGCSE)*, pages 141–145, 1989. DOI: <https://www.doi.org/10.1145/65293.71203>.
- [144] M. Jump and K. S. McKinley. Cork: Dynamic Memory Leak Detection for Garbage-collected Languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 31–38, 2007. DOI: <https://www.doi.org/10.1145/1190216.1190224>.
- [145] M. Jump and K. S. McKinley. Dynamic Shape Analysis via Degree Metrics. In *Proceedings of the 2009 International Symposium on Memory Management (ISMM)*, pages 119–128, 2009. DOI: <https://www.doi.org/10.1145/1542431.1542449>.
- [146] M. Jump and K. S. McKinley. Detecting Memory Leaks in Managed Languages with Cork. *Softw. Pract. Exp.*, 40(1):1–22, 2010. DOI: <https://www.doi.org/10.1002/spe.945>.
- [147] R. Jung and M. Adolf. The JPetStore Suite: A Concise Experiment Setup for Research. In *Proceedings of the 9th Symposium on Software Performance (SSP)*, 2018. URL: <http://eprints.uni-kiel.de/48775/>.
- [148] R. Jung, M. Adolf, and C. Dornieden. Towards Extracting Realistic User Behavior Models. *Softwaretechnik-Trends*, 37(3), November 2017. URL: <http://eprints.uni-kiel.de/40365/>.
- [149] T. Kamada and S. Kawai. A General Framework for Visualizing Abstract Objects and Relations. *ACM Trans. Graph.*, 10(1):1–39, 1991. DOI: <https://www.doi.org/10.1145/99902.99903>.
- [150] S. Kelley, E. Aftandilian, C. Gramazio, N. P. Ricci, S. L. Su, and S. Z. Guyer. Heapviz: Interactive Heap Visualization for Program Understanding and Debugging. *Information Visualization*, 12(2):163–177, 2013. DOI: <https://www.doi.org/10.1177/1473871612438786>.

- [151] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001. DOI: https://www.doi.org/10.1007/3-540-45337-7_18.
- [152] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting started with ASPECTJ. *Commun. ACM*, 44(10):59–65, 2001. DOI: <https://www.doi.org/10.1145/383845.383858>.
- [153] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, 1997. DOI: <https://www.doi.org/10.1007/BFb0053381>.
- [154] Kieker Project. Kieker, 2021. URL: <http://kieker-monitoring.net/>. last visited on 2021-04-08.
- [155] R. M. Kitchin. Cognitive Maps: What are they and why study them? *Journal of Environmental Psychology*, 14(1):1–19, 1994. DOI: [https://www.doi.org/10.1016/S0272-4944\(05\)80194-X](https://www.doi.org/10.1016/S0272-4944(05)80194-X).
- [156] C. Knight and M. Munro. Comprehension with[in] Virtual Environment Visualisations. In *Proceedings of the 7th International Workshop on Program Comprehension (IWPC)*, pages 4–11, 1999. DOI: <https://www.doi.org/10.1109/WPC.1999.777733>.
- [157] C. Knight and M. Munro. Virtual but Visible Software. In *Proceedings of the International Conference on Information Visualisation (IV)*, pages 198–205, 2000. DOI: <https://www.doi.org/10.1109/IV.2000.859756>.
- [158] D. E. Knuth. Top-Down Syntax Analysis. *Acta Informatica*, 1:79–110, 1971. DOI: <https://www.doi.org/10.1007/BF00289517>.
- [159] A. J. Ko, T. D. LaToza, and M. M. Burnett. A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants. *Empirical Software Engineering*, 20(1):110–141, 2015. DOI: <https://www.doi.org/10.1007/s10664-013-9279-3>.
- [160] M. Koller and G. Meixner. Task Models in Practice: Are There Special Requirements for the Use in Daily Work? In *Proceedings of 18th International Conference on Human-Computer Interaction (HCI) - Theory, Design, Development and Practice*, pages 488–497, 2016. DOI: https://www.doi.org/10.1007/978-3-319-39510-4_45.
- [161] S. Kristoffersen. Learnability and Robustness of User Interfaces. Towards a Formal Analysis of Usability Design Principles. In *Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT)*, Volume SE/MUSE/GSDCA, pages 261–268, 2008.
- [162] L. M. Kritzinger, T. Krismayer, R. Rabiser, and P. Grünbacher. A User Study on the Usefulness of Visualization Support for Requirements Monitoring. In *Proceedings of the 7th IEEE Working Conference on Software Visualization (VISSOFT)*, pages 56–66, 2019. DOI: <https://www.doi.org/10.1109/VISSOFT.2019.00015>.
- [163] G. Kruk, O. D. S. Alves, and L. Molinari. JavaFX Charts: Implementation of Missing Features. In *Proceedings of the International Conference on Accelerator and Large Experimental Control Systems (ICALEPCS)*, number 16, pages 866–868, Jan. 2017. DOI: <https://www.doi.org/10.18429/JACoW-ICALEPCS2017-TUPHA186>.
- [164] J. B. Kruskal and J. M. Landwehr. Icicle Plots: Better Displays for Hierarchical Clustering. *The American Statistician*, 37(2):162–168, 1983. DOI: <https://www.doi.org/10.2307/2685881>.
- [165] E. Kuleshov. Using the ASM Framework to Implement Common Java Bytecode Transformation Patterns, 2007.
- [166] M. Kutar, C. Britton, and J. Wilson. Cognitive Dimensions: An Experience Report. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group (PPIG)*, page 7, 2000. URL: <http://ppig.org/library/paper/cognitive-dimensions-experience-report>.
- [167] M. Kutar, C. L. Nehaniv, C. Britton, and S. Jones. The Cognitive Dimensions of an Artifact vis-à-vis Individual Human Users: Studies with Notations for the Temporal Specification of Interactive Systems. In *Proceedings of the 4th International Conference on Cognitive Technology*, pages 342–355, 2001. DOI: https://www.doi.org/10.1007/3-540-44617-6_32.
- [168] G. Lakoff. *Master Metaphor List*. University of California, 1994.
- [169] G. Langelier and K. Dhambri. Visual Analysis of Azureus using VERSO. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 163–164, 2007. DOI: <https://www.doi.org/10.1109/VISSOFT.2007.4290720>.
- [170] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based Analysis of Quality for Large-scale Software Systems. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 214–223, 2005. DOI: <https://www.doi.org/10.1145/1101908.1101941>.

- [171] G. Langelier, H. A. Sahraoui, and P. Poulin. Exploring the Evolution of Software Quality with Animated Visualization. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 13–20, 2008. DOI: <https://www.doi.org/10.1109/VLHCC.2008.4639052>.
- [172] P. Laporte. JVM Options - the complete reference, 2021. URL: <http://pingtimeout.github.io/jvm-options/>. last visited on 2021-06-15.
- [173] P. Lengauer, V. Bitto, S. Fitzek, M. Weninger, and H. Mössenböck. Efficient Memory Traces with Full Pointer Information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*, pages 4:1–4:11, 2016. DOI: <https://www.doi.org/10.1145/2972206.2972220>.
- [174] P. Lengauer, V. Bitto, and H. Mössenböck. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 51–62, 2015. DOI: <https://www.doi.org/10.1145/2668930.2688037>.
- [175] P. Lengauer, V. Bitto, and H. Mössenböck. Efficient and Viable Handling of Large Object Traces. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 249–260, 2016. DOI: <https://www.doi.org/10.1145/2851553.2851555>.
- [176] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 3–14, 2017. DOI: <https://www.doi.org/10.1145/3030207.3030211>.
- [177] T. Lengauer and R. E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979. DOI: <https://www.doi.org/10.1145/357062.357071>.
- [178] B. Li, J. A. Burgoyne, and I. Fujinaga. Extending Audacity for Audio Annotation. In *Proceedings of the 7th International Conference on Music Information Retrieval (ISMIR)*, pages 379–380, 2006.
- [179] J. Li, H. Izakian, W. Pedrycz, and I. Jamal. Clustering-based Anomaly Detection in Multivariate Time Series Data. *Appl. Soft Comput.*, 100:106919, 2021. DOI: <https://www.doi.org/10.1016/j.asoc.2020.106919>.
- [180] J. Liang and M. L. Huang. Highlighting in Information Visualization: A Survey. In *Proceedings of the 14th International Conference on Information Visualisation (IV)*, pages 79–85. IEEE Computer Society, 2010. DOI: <https://www.doi.org/10.1109/IV.2010.21>.
- [181] P. Lidén and S. Karlsson. The Z Garbage Collector - An Introduction, FOSDEM 2018, 2018. URL: <http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf>. last visited on 2021-04-08.
- [182] H. Lieberman and C. Hewitt. A Real-time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM*, 26(6):419–429, 1983. DOI: <https://www.doi.org/10.1145/358141.358147>.
- [183] D. Limberger, W. Scheibel, J. Dollner, and M. Trapp. Advanced Visual Metaphors and Techniques for Software Maps. In *Proceedings of the 12th International Symposium on Visual Information Communication and Interaction (VINCI)*, pages 11:1–11:8, 2019. DOI: <https://www.doi.org/10.1145/3356422.3356444>.
- [184] Q. Limbourg and J. Vanderdonckt. *The Handbook of Task Analysis for Human-Computer Interaction*, chapter Comparing Task Models for User Interface Design, pages 135–154. CRC Press, 2003.
- [185] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014. DOI: <https://www.doi.org/10.1109/TVCG.2014.2346452>.
- [186] V. López-Jaquero and F. M. Simarro. Comprehensive Task and Dialog Modelling. In *Proceedings of the 12th International Conference on Human-Computer Interaction (HCI) - Interaction Design and Usability*, volume 4550, pages 1149–1158. Springer, 2007. DOI: https://www.doi.org/10.1007/978-3-540-73105-4_125.
- [187] J. P. Magalhães and L. M. Silva. Adaptive Monitoring of Web-based Applications: A Performance Study. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, pages 471–478, 2013. DOI: <https://www.doi.org/10.1145/2480362.2480454>.
- [188] L. Makor. Using Tree Visualizations to Facilitate Memory Leak Analysis. Master's thesis, Johannes Kepler University, Institute for System Software, 2020. URL: <https://epub.jku.at/obvulihs/content/titleinfo/5473677>.
- [189] D. Mapelsden, J. Hosking, and J. Grundy. Design Pattern Modelling and Instantiation Using DPML. In *Proceedings of the 14th International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT)*, pages 3–11, 2002. URL: <http://dl.acm.org/citation.cfm?id=564092.564094>.

- [190] D. Marinov and R. O’Callahan. Object Equality Profiling. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 313–325, 2003. DOI: <https://www.doi.org/10.1145/949305.949333>.
- [191] M. Marron, C. Sánchez, Z. Su, and M. Fähndrich. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Software Eng.*, 39(6):774–786, 2013. DOI: <https://www.doi.org/10.1109/TSE.2012.69>.
- [192] E. K. Maxwell, G. Back, and N. Ramakrishnan. Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In B. Rao, B. Krishnapuram, A. Tomkins, and Q. Yang, editors, *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 115–124. ACM, 2010. DOI: <https://www.doi.org/10.1145/1835804.1835822>.
- [193] M. T. Maybury. Intelligent User Interfaces: An Introduction. In *Proceedings of the 4th International Conference on Intelligent User Interfaces (IUI)*, pages 3–4, 1999. DOI: <https://www.doi.org/10.1145/291080.291081>.
- [194] K. L. McGraw and B. A. McGraw. Wizards, Coaches, Advisors, and More: A Performance Support Primer. In *Extended Abstracts on Human Factors in Computing Systems (CHI)*, pages 152–153, 1997. DOI: <https://www.doi.org/10.1145/1120212.1120318>.
- [195] G. Meixner, M. Seissler, and K. Breiner. Model-Driven Useware Engineering. In *Model-Driven Development of Advanced User Interfaces*, pages 1–26. 2011. DOI: https://www.doi.org/10.1007/978-3-642-14562-9_1.
- [196] L. Merino, M. Hess, A. Bergel, O. Nierstrasz, and D. Weiskopf. PerfVis: Pervasive Visualization in Immersive Augmented Reality for Performance Awareness. In *Companion Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 13–16, 2019. DOI: <https://www.doi.org/10.1145/3302541.3313104>.
- [197] N. Mitchell. The Runtime Structure of Object Ownership. In *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)*, pages 74–98, 2006. DOI: https://www.doi.org/10.1007/11785477_5.
- [198] N. Mitchell, E. Schonberg, and G. Sevitsky. Making Sense of Large Heaps. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009. DOI: https://www.doi.org/10.1007/978-3-642-03013-0_5.
- [199] N. Mitchell, E. Schonberg, and G. Sevitsky. Four Trends Leading to Java Runtime Bloat. *IEEE Software*, 27(1):56–63, 2010. DOI: <https://www.doi.org/10.1109/MS.2010.7>.
- [200] N. Mitchell and G. Sevitsky. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In L. Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743, pages 351–377, 2003. DOI: https://www.doi.org/10.1007/978-3-540-45070-2_16.
- [201] N. Mitchell and G. Sevitsky. The Causes of Bloat, the Limits of Health. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007. DOI: <https://www.doi.org/10.1145/1297027.1297046>.
- [202] N. Mitchell, G. Sevitsky, P. Kumanan, and E. Schonberg. Data Structure Health. In *Fifth International Workshop on Dynamic Analysis (WODA@ICSE)*, page 2, 2007. DOI: <https://www.doi.org/10.1109/WODA.2007.1>.
- [203] H. Mössenböck, M. Löberbauer, and A. Wöß. The Compiler Generator Coco/R, 2018. URL: <http://www.ssw.uni-linz.ac.at/Coco/>. last visited on 2021-04-08.
- [204] S. Murray. *Interactive Data Visualization for the Web*. O’Reilly Media, 2013.
- [205] MyBatis. JPetStore, 2016. URL: <http://mybatis.org/jpetstore-6/>. last visited on 2021-04-08.
- [206] R. H. Myers and R. H. Myers. *Classical and Modern Regression with Applications*, volume 2. Duxbury Press Belmont, 1990.
- [207] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.*, 89(2):44–66, 2003. DOI: [https://www.doi.org/10.1016/S1571-0661\(04\)81042-9](https://www.doi.org/10.1016/S1571-0661(04)81042-9).
- [208] J. Nielsen. *Usability Engineering*. Academic Press, 1993.
- [209] M. Nørgaard and K. Hornbæk. What Do Usability Evaluators Do in Practice? An Explorative Study of Think-Aloud Testing. In *Proceedings of the Conference on Designing Interactive Systems*, pages 209–218, 2006. DOI: <https://www.doi.org/10.1145/1142405.1142439>.
- [210] K. Ogami, R. G. Kula, H. Hata, T. Ishio, and K. Matsumoto. Using High-Rising Cities to Visualize Performance in Real-Time. In *Proceedings of the IEEE Working Conference on Software Visualization (VISSOFT)*, pages 33–42, 2017. DOI: <https://www.doi.org/10.1109/VISSOFT.2017.25>.

- [211] Oracle. JVM Tool Interface Version 1.2.3, 2013. URL: <https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>. last visited on 2021-04-08.
- [212] Oracle. Java Flight Recorder, 2014. URL: <https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170>. last visited on 2021-04-08.
- [213] Oracle. jmap, 2018. URL: <https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html>. last visited on 2021-04-08.
- [214] Oracle. ZGC - The Z Garbage Collector, 2019. URL: <http://openjdk.java.net/projects/zgc/>. last visited on 2021-04-08.
- [215] Oracle. HPROF: A Heap/CPU Profiling Tool, 2020. URL: <https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html>. last visited on 2021-04-08.
- [216] Oracle. Java Mission Control, 2020. URL: <https://openjdk.java.net/projects/jmc/>. last visited on 2021-04-08.
- [217] Oracle. JConsole, 2020. URL: <https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html>. last visited on 2021-04-08.
- [218] Oracle. jhat, 2021. URL: <https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jhat.html>. last visited on 2021-04-08.
- [219] Oracle. The HotSpot Group, 2021. URL: <http://openjdk.java.net/groups/hotspot/>. last visited on 2021-04-08.
- [220] Oracle. VisualVM, 2021. URL: <https://unity.com>. last visited on 2021-04-08.
- [221] Oracle. VisualVM: All-in-One Java Troubleshooting Tool, 2021. URL: <https://visualvm.github.io/>. last visited on 2021-04-08.
- [222] J. D. Ornelas, J. C. Silva, and J. L. Silva. USS: User Support System. In *Proceedings of the 11th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2016. DOI: <https://www.doi.org/10.1109/CISTI.2016.7521412>.
- [223] K. O’Hair. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network, Developer Technical Articles & Tips*, 28, 2004.
- [224] T. Panas, R. Berrigan, and J. C. Grundy. A 3D Metaphor for Software Production Visualization. In *Proceedings of the Seventh International Conference on Information Visualization (IV)*, pages 314–319, 2003. DOI: <https://www.doi.org/10.1109/IV.2003.1217996>.
- [225] F. Paternò, C. Mancini, and S. Meniconi. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In *Proceedings of the IFIP TC13 International Conference on Human-Computer Interaction (INTERACT)*, pages 362–369, 1997.
- [226] W. D. Pauw and G. Sevitsky. Visualizing Reference Patterns for Solving Memory Leaks in Java. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, pages 116–134, 1999. DOI: https://www.doi.org/10.1007/3-540-48743-3_6.
- [227] M. Peiris and J. H. Hill. Automatically Detecting ”Excessive Dynamic Memory Allocations” Software Performance Anti-Pattern. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 237–248, 2016. DOI: <https://www.doi.org/10.1145/2851553.2851563>.
- [228] J. Potter, J. Noble, and D. G. Clarke. The Ins and Outs of Objects. In *Proceedings of the Australian Software Engineering Conference (ASWEC)*, pages 80–89, 1998. DOI: <https://www.doi.org/10.1109/ASWEC.1998.730915>.
- [229] T. Printezis and R. Jones. GCspy: An Adaptable Heap Visualisation Framework. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 343–358, 2002. DOI: <https://www.doi.org/10.1145/582419.582451>.
- [230] A. Prokopec, A. Rosà, D. Leopoldseeder, G. Duboscq, P. Tuma, M. Studener, L. Bulej, Y. Zheng, A. Villazón, D. Simon, T. Würthinger, and W. Binder. Renaissance: benchmarking suite for parallel applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 31–47, 2019. DOI: <https://www.doi.org/10.1145/3314221.3314637>.
- [231] J. Qian and X. Zhou. Inferring Weak References for Fixing Java Memory Leaks. In *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 571–574, 2012. DOI: <https://www.doi.org/10.1109/ICSM.2012.6405323>.

- [232] R. Rabiser, P. Grünbacher, and M. Lehofer. A Qualitative Study on User Guidance Capabilities in Product Configuration Tools. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 110–119, 2012. DOI: <https://www.doi.org/10.1145/2351676.2351693>.
- [233] R. Rabiser, S. Guinea, M. Vierhauser, L. Baresi, and P. Grünbacher. A Comparison Framework for Runtime Monitoring Approaches. *Journal of Systems and Software*, 125:309–321, 2017. DOI: <https://www.doi.org/10.1016/j.jss.2016.12.034>.
- [234] R. Rabiser, K. Schmid, H. Eichelberger, M. Vierhauser, S. Guinea, and P. Grünbacher. A Domain Analysis of Resource and Requirements Monitoring: Towards a Comprehensive Model of the Software Monitoring Domain. *Information & Software Technology*, 111:86–109, 2019. DOI: <https://www.doi.org/10.1016/j.infsof.2019.03.013>.
- [235] R. Rabiser, M. Vierhauser, and P. Grünbacher. Assessing the Usefulness of a Requirements Monitoring Tool: A Study Involving Industrial Software Engineers. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 122–131, 2016. DOI: <https://www.doi.org/10.1145/2889160.2889234>.
- [236] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik. User-Guided Program Reasoning Using Bayesian Inference. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 722–735, New York, NY, USA, 2018. DOI: <https://www.doi.org/10.1145/3192366.3192417>.
- [237] G. Ramalingam and T. Reps. An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 287–296, 1994. DOI: <https://www.doi.org/10.1145/174675.177905>.
- [238] D. Rayside and L. Mendel. Object Ownership Profiling: A Technique for Finding and Fixing Memory Leaks. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 194–203, 2007. DOI: <https://www.doi.org/10.1145/1321631.1321661>.
- [239] D. Rayside, L. Mendel, and D. Jackson. A Dynamic Analysis for Revealing Object Ownership and Sharing. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis (WODA)*, pages 57–64, 2006. DOI: <https://www.doi.org/10.1145/1138912.1138924>.
- [240] S. Raza and C. Ding. Progress in Context-aware Recommender Systems - An Overview. *Comput. Sci. Rev.*, 31:84–97, 2019. DOI: <https://www.doi.org/10.1016/j.cosrev.2019.01.001>.
- [241] S. P. Reiss. Visualizing the Java Heap. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM)*, pages 389–390, 2009. DOI: <https://www.doi.org/10.1109/ICSM.2009.5306287>.
- [242] S. P. Reiss. Visualizing the Java Heap to Detect Memory Problems. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 73–80, 2009. DOI: <https://www.doi.org/10.1109/VISSOFT.2009.5336418>.
- [243] S. P. Reiss. Visualizing the Java Heap. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 251–254, 2010. DOI: <https://www.doi.org/10.1145/1810295.1810344>.
- [244] B. Reitinger, D. Kranzlmüller, and A. Ferko. Program Visualization Through Visual Metaphors. In *Proceedings of the International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2003. URL: http://wscg.zcu.cz/wscg2003/Papers_2003/J79.pdf.
- [245] B. Reitinger, D. Kranzlmüller, and J. Volkert. The MOST Immersive Approach for Parallel and Distributed Program Analysis. In *Proceedings of the International Conference on Information Visualisation (IV)*, pages 517–522, 2001. DOI: <https://www.doi.org/10.1109/IV.2001.942105>.
- [246] J. Renz, T. Staubitz, J. Pollak, and C. Meinel. Improving the Onboarding User Experience in MOOCs. In *Proceedings of the 6th International Conference on Education and New Learning Technologies (EDULEARN)*, pages 3931–3941, 7–9 July, 2014 2014. URL: https://hpi.de/fileadmin/user_upload/fachgebiete/meinel/papers/Web-University/2014_Renz_EDULEARN.pdf.
- [247] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Generating Program Traces with Object Death Records. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 139–142, 2011. DOI: <https://www.doi.org/10.1145/2093157.2093178>.
- [248] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Portable Production of Complete and Precise GG Traces. In *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 109–118, 2013. DOI: <https://www.doi.org/10.1145/2464157.2466484>.

- [249] C. K. Riemenschneider and B. C. Hardgrave. Explaining Software Development Tool Use with the Technology Acceptance Model. *Journal of Computer Information Systems (JCIS)*, 41(4):1–8, 2001. URL: <https://www.tandfonline.com/doi/abs/10.1080/08874417.2001.11647015>.
- [250] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza. On The Use of Virtual Reality in Software Visualization: The Case of the City Metaphor. *Inf. Softw. Technol.*, 114:92–106, 2019. DOI: <https://www.doi.org/10.1016/j.infsof.2019.06.007>.
- [251] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza. The City Metaphor in Software Visualization: Feelings, Emotions, and Thinking. *Multim. Tools Appl.*, 78(23):33113–33149, 2019. DOI: <https://www.doi.org/10.1007/s11042-019-07748-1>.
- [252] C. Ruggieri and T. P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 285–293, 1988. DOI: <https://www.doi.org/10.1145/73560.73585>.
- [253] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. DOI: <https://www.doi.org/10.1007/s10664-008-9102-8>.
- [254] S. Saito. ProcessCity - Visualizing Business Processes as City Metaphor. In *Proceedings of the CAiSE Forum on Information Systems Engineering in Responsible Information Systems*, pages 207–214, 2019. DOI: https://www.doi.org/10.1007/978-3-030-21297-1_18.
- [255] A. Savidis and N. Koutsopoulos. Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features. In *Proceedings of the 7th International Symposium on Advances in Visual Computing (ISVC)*, pages 259–268, 2011. DOI: https://www.doi.org/10.1007/978-3-642-24028-7_24.
- [256] R. C. Schank, T. R. Berman, and K. A. Macpherson. Learning by Doing. *Instructional-design Theories and Models: A New Paradigm of Instructional Theory*, 2(2):161–181, 1999.
- [257] W. Scheibel, M. Trapp, D. Limberger, and J. Döllner. A Taxonomy of Treemap Visualization Techniques. In *Proceedings of the 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, pages 273–280, 2020. DOI: <https://www.doi.org/10.5220/0009153902730280>.
- [258] W. Scheibel, C. Weyand, and J. Döllner. EvoCells - A Treemap Layout Algorithm for Evolving Tree Data. In *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP)*, pages 273–280, 2018. DOI: <https://www.doi.org/10.5220/0006617102730280>.
- [259] A. Schörgenhumer, P. Hofer, D. Gnedt, and H. Mössenböck. Efficient Sampling-based Lock Contention Profiling for Java. In *Proceedings of the 8th ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 331–334, 2017. DOI: <https://www.doi.org/10.1145/3030207.3030234>.
- [260] A. Schörgenhumer, M. Kahlhofer, P. Grünbacher, and H. Mössenböck. Can we Predict Performance Events with Time Series Data from Monitoring Multiple Systems? In *Companion of the ACM/SPEC International Conference on Performance Engineering ICPE*, pages 9–12, 2019. DOI: <https://www.doi.org/10.1145/3302541.3313101>.
- [261] H. Schulz. Treevis.net: A Tree Visualization Reference. *IEEE Computer Graphics and Applications*, 31(6):11–15, 2011. DOI: <https://www.doi.org/10.1109/MCG.2011.103>.
- [262] H. Schulz and H. Schumann. Visualizing Graphs - A Generalized View. In *Proceedings of the 10th International Conference on Information Visualisation (IV)*, pages 166–173, 2006. DOI: <https://www.doi.org/10.1109/IV.2006.130>.
- [263] A. Schörgenhumer. Efficient Sampling-based Lock Contention Profiling in Java. Master’s thesis, Johannes Kepler University, Institute for System Software, 2017. URL: <https://epub.jku.at/obvulihs/content/titleinfo/1825350>.
- [264] C. B. Seaman. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.*, 25(4):557–572, 1999. DOI: <https://www.doi.org/10.1109/32.799955>.
- [265] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 657–676, 2011. DOI: <https://www.doi.org/10.1145/2048066.2048118>.
- [266] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap Profiling for Space-efficient Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 104–113, 2001. DOI: <https://www.doi.org/10.1145/378795.378820>.


- [267] R. Shaham, E. K. Kolodner, and S. Sagiv. Automatic Removal of Array Memory Leaks in Java. In *Proceedings of the 9th International Conference on Compiler Construction*, volume 1781, pages 50–66, 2000. DOI: https://www.doi.org/10.1007/3-540-46423-9_4.
- [268] B. Shneiderman. Tree Visualization with Tree-Maps: 2-D Space-Filling Approach. *ACM Trans. Graph.*, 11(1):92–99, 1992. DOI: <https://www.doi.org/10.1145/102377.115768>.
- [269] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the IEEE Symposium on Visual Languages (VL)*, pages 336–343, 1996. DOI: <https://www.doi.org/10.1109/VL.1996.545307>.
- [270] C. U. Smith and L. G. Williams. Software Performance Antipatterns. In *Proceedings of the International Workshop on Software and Performance (WOSP)*, pages 127–136, 2000. DOI: <https://www.doi.org/10.1145/350391.350420>.
- [271] C. U. Smith and L. G. Williams. New Software Performance AntiPatterns: More Ways to Shoot Yourself in the Foot. In *Proceedings of the 28th International Computer Measurement Group Conference*, pages 667–674, 2002.
- [272] C. U. Smith and L. G. Williams. More New Software Antipatterns: Even More Ways to Shoot Yourself in the Foot. In *Proceedings of the 29th International Computer Measurement Group Conference*, pages 717–725, 2003.
- [273] M. Sondag, B. Speckmann, and K. Verbeek. Stable Treemaps via Local Moves. *IEEE Trans. Vis. Comput. Graph.*, 24(1):729–738, 2018. DOI: <https://www.doi.org/10.1109/TVCG.2017.2745140>.
- [274] K. Soong, X. Fu, and Y. Zhou. Optimizing New User Experience in Online Services. In *Proceedings of the 5th IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 442–449, 2018. DOI: <https://www.doi.org/10.1109/DSAA.2018.00057>.
- [275] V. Sor, P. Ou, T. Treier, and S. N. Srirama. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 544–547, 2013. DOI: <https://www.doi.org/10.1109/ICSM.2013.92>.
- [276] V. Sor, N. Salnikov-Tarnowski, and S. N. Srirama. Automated Statistical Approach for Memory Leak Detection: Case Studies. In *Proceedings of the International Conferences On the Move to Meaningful Internet Systems (OTM)*, volume 7045, pages 635–642, 2011. DOI: https://www.doi.org/10.1007/978-3-642-25106-1_16.
- [277] V. Sor and S. N. Srirama. A Statistical Approach for Identifying Memory Leaks in Cloud Applications. In *Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER)*, pages 623–628, 2011.
- [278] V. Sor and S. N. Srirama. Memory Leak Detection in Java: Taxonomy and Classification of Approaches. *J. Syst. Softw.*, 96:139–151, 2014. DOI: <https://www.doi.org/10.1016/j.jss.2014.06.005>.
- [279] A. Sosnówka. Test City Metaphor as Support for Visual Testcase Analysis Within Integration Test Domain. In *Proceedings of the Federated Conference on Computer Science and Information Systems*, pages 1353–1358, 2013. URL: <http://ieeexplore.ieee.org/document/6644194/>.
- [280] A. Sosnówka. Test City Metaphor for Low Level Tests Restructuration in Test Database. In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 141–150, 2013. DOI: https://www.doi.org/10.1007/978-3-642-54092-9_10.
- [281] A. Sosnówka. Testware Visualized - Visual Support for Testware Reorganization. In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pages 109–114, 2013. DOI: <https://www.doi.org/10.5220/0004451001090114>.
- [282] M. Staron, W. Meding, J. Hansson, C. Höglund, K. Niesel, and V. Bergmann. Dashboards for Continuous Monitoring of Quality for Software Product under Development. In *Relating System Quality and Software Architecture*, pages 209–229, 2014. DOI: <https://www.doi.org/10.1016/b978-0-12-417009-4.00008-9>.
- [283] J. T. Stasko and E. Zhang. Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations. In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS)*, pages 57–65, 2000. DOI: <https://www.doi.org/10.1109/INFVIS.2000.885091>.
- [284] F. Steinbrückner. Coherent Software Cities. In *Proceedings of the 26th IEEE International Conference on Software Maintenance (ICSM)*, pages 1–2, 2010. DOI: <https://www.doi.org/10.1109/ICSM.2010.5610421>.
- [285] F. Steinbrückner and C. Lewerentz. Representing Development History in Software Cities. In *Proceedings of the ACM Symposium on Software Visualization (SOFTVIS)*, pages 193–202, 2010. DOI: <https://www.doi.org/10.1145/1879211.1879239>.
- [286] F. Steinbrückner and C. Lewerentz. Understanding Software Evolution with Software Cities. *Information Visualization*, 12(2):200–216, 2013. DOI: <https://www.doi.org/10.1177/1473871612438785>.

- [287] P. N. Sukaviriya and J. D. Foley. Coupling a UI Framework with Automatic Generation of Context-Sensitive Animated Help. In *Proceedings of the 3rd Annual ACM Symposium on User Interface Software and Technology (UIST)*, pages 152–166, 1990. DOI: <https://www.doi.org/10.1145/97924.97942>.
- [288] C. Szabo. Novice Code Understanding Strategies during a Software Maintenance Assignment. In *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 276–284, 2015. DOI: <https://www.doi.org/10.1109/ICSE.2015.341>.
- [289] Y. Tang, Q. Gao, and F. Qin. LeakSurvivor: Towards Safely Tolerating Memory Leaks for Garbage-Collected Languages. In *Proceedings of the USENIX Annual Technical Conference*, pages 307–320, 2008. URL: http://www.usenix.org/events/usenix08/tech/full_papers/tang/tang.pdf.
- [290] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting Sequence Diagram from Execution Trace of Java Program. In *Proceedings of the 8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 148–154, 2005. DOI: <https://www.doi.org/10.1109/IWPSE.2005.19>.
- [291] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms (Working Paper). In *Proc. of the 12th Annual Symposium on Switching and Automata Theory (SWAT)*, pages 114–121, 1971. DOI: <https://www.doi.org/10.1109/SWAT.1971.10>.
- [292] R. E. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. DOI: <https://www.doi.org/10.1137/0201010>.
- [293] Technische Universität Darmstadt. Scala Benchmark Suite, 2012. URL: <http://www.benchmarks.scalabench.org/modules/scala-benchmark-suite/>. last visited on 2021-04-08.
- [294] S. T. Teoh. A Study on Multiple Views for Tree Visualization. In *Proceedings of SPIE - Visualization and Data Analysis*, volume 6495, 2007. DOI: <https://www.doi.org/10.1117/12.703076>.
- [295] The Standard Performance Evaluation Corporation (SPEC). SPECjvm2008, 2021. URL: <https://www.spec.org/jvm2008/>. last visited on 2021-04-08.
- [296] The Valgrind Developers. Valgrind, 2021. URL: <http://valgrind.org/>. last visited on 2021-04-08.
- [297] D. Tidwell and J. Fuccella. TaskGuides: Instant Wizards on the Web. In *Proceedings of the 15th Annual International Conference on Computer Documentation SIGDOC*, pages 263–272, 1997. DOI: <https://www.doi.org/10.1145/263367.263401>.
- [298] M. Tory and T. Möller. Human Factors in Visualization Research. *IEEE Trans. Vis. Comput. Graph.*, 10(1):72–84, 2004. DOI: <https://www.doi.org/10.1109/TVCG.2004.1260759>.
- [299] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin. ALETHEIA: Improving the Usability of Static Security Analysis. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 762–774, 2014. DOI: <https://www.doi.org/10.1145/2660267.2660339>.
- [300] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1992.
- [301] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, pages 157–167, 1984. DOI: <https://www.doi.org/10.1145/800020.808261>.
- [302] S. van den Elzen and J. J. van Wijk. Small Multiples, Large Singles: A New Approach for Visual Data Exploration. *Comput. Graph. Forum*, 32(3):191–200, 2013. DOI: <https://www.doi.org/10.1111/cgf.12106>.
- [303] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous Monitoring of Software Services: Design and Application of the Kieker Framework. Technical Report TR-0921, Department of Computer Science, Kiel University, Germany, Nov. 2009.
- [304] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the Third Joint WOSP/SIPEW International Conference on Performance Engineering (ICPE)*, pages 247–248, 2012. DOI: <https://www.doi.org/10.1145/2188286.2188326>.
- [305] R. L. Veroy and S. Z. Guyer. Garbology: A Study of How Java Objects Die. In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, pages 168–179, 2017. DOI: <https://www.doi.org/10.1145/3133850.3133854>.
- [306] J. Waller, C. Wulf, F. Fittkau, P. Dohring, and W. Hasselbring. Synchrovis: 3D Visualization of Monitoring Traces in the City Metaphor for Analyzing Concurrency. In *Proceedings of the 1st IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013. DOI: <https://www.doi.org/10.1109/VISSOFT.2013.6650520>.

- [307] J. Wang, X. Peng, Z. Xing, and W. Zhao. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *Proceedings of the 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 213–222, 2011. DOI: <https://www.doi.org/10.1109/ICSM.2011.6080788>.
- [308] Y. Wang, S. T. Teoh, and K. Ma. Evaluating the Effectiveness of Tree Visualization Systems for Knowledge Discovery. In *Proceedings of the Joint Eurographics - IEEE VGTC Symposium on Visualization (EuroVis)*, pages 67–74, 2006. DOI: <https://www.doi.org/10.2312/VisSym/EuroVis06/067-074>.
- [309] M. O. Ward, G. G. Grinstein, and D. A. Keim. *Interactive Data Visualization - Foundations, Techniques, and Applications*. A K Peters, 2010. URL: <http://www.akpeters.com/product.asp?ProdCode=4735>.
- [310] C. Ware. Chapter One - Foundations for an Applied Science of Data Visualization. In *Information Visualization (Third Edition)*, Interactive Technologies, pages 1 – 30. 2013. DOI: <https://www.doi.org/https://doi.org/10.1016/B978-0-12-381464-7.00001-6>.
- [311] M. Weber, M. Alexa, and W. Müller. Visualizing Time-Series on Spirals. In K. Andrews, S. F. Roth, and P. C. Wong, editors, *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS)*, pages 7–14, 2001. DOI: <https://www.doi.org/10.1109/INFVIS.2001.963273>.
- [312] P. Wegner and E. D. Reilly. Data Structures. In *Encyclopedia of Computer Science*, pages 507–512. URL: <http://dl.acm.org/citation.cfm?id=1074100.1074312>.
- [313] Q. Wen, J. Gao, X. Song, L. Sun, H. Xu, and S. Zhu. RobustSTL: A Robust Seasonal-Trend Decomposition Algorithm for Long Time Series. In *Proceedings of the Thirty-Third Conference on Artificial Intelligence (AAAI)*, pages 5409–5416, 2019. DOI: <https://www.doi.org/10.1609/aaai.v33i01.33015409>.
- [314] M. Weninger et al. AntTracks - Memory Monitoring using Accurate and Efficient Object Tracing for Java Applications, 2021. URL: <http://mevss.jku.at/AntTracks>. last visited on 2021-04-08.
- [315] M. Weninger, E. Gander, and H. Mössenböck. Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools. *Proc. ACM Hum.-Comput. Interact.*, 5(EICS), June 2021. DOI: <https://www.doi.org/10.1145/3461731>.
- [316] M. Weninger, E. Gander, and H. Mössenböck. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proceedings of the 9th Symposium on Software Performance (SSP)*, pages 64–66, 2018. URL: http://pi.informatik.uni-siegen.de/stt/39_3/01_Fachgruppenberichte/SSP18/WeningerGanderMoessenboeck18.pdf.
- [317] M. Weninger, E. Gander, and H. Mössenböck. Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes (ManLang)*, pages 14:1–14:13, 2018. DOI: <https://www.doi.org/10.1145/3237009.3237023>.
- [318] M. Weninger, E. Gander, and H. Mössenböck. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 273–284, 2019. DOI: <https://www.doi.org/10.1145/3297663.3310297>.
- [319] M. Weninger, E. Gander, and H. Mössenböck. Detection of Suspicious Time Windows In Memory Monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*, pages 95–104, 2019. DOI: <https://www.doi.org/10.1145/3357390.3361025>.
- [320] M. Weninger, E. Ganer, and H. Mössenböck. Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance. In *Proceedings of the 11th Symposium on Software Performance (SSP)*, 2020. URL: https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2020/Papers/SSP2020_paper_7.pdf.
- [321] M. Weninger, P. Grünbacher, E. Gander, and A. Schörgenhuber. Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proceedings ACM Hum.-Comput. Interact.*, 4(EICS), June 2020. DOI: <https://www.doi.org/10.1145/3394977>.
- [322] M. Weninger, P. Grünbacher, H. Zhang, T. Yue, and S. Ali. Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment. In *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 21–30, 2018. DOI: <https://www.doi.org/10.1109/APSEC.2018.00016>.
- [323] M. Weninger, P. Lengauer, and H. Mössenböck. User-centered Offline Analysis of Memory Monitoring Data. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*, pages 357–360, 2017. DOI: <https://www.doi.org/10.1145/3030207.3030236>.
- [324] M. Weninger and L. Makor. HttpClient Leak Driver, 2020. URL: <https://github.com/NeonMika/httpclient-leak-driver/>. last visited on 2021-04-08.

- [325] M. Weninger, L. Makor, E. Gander, and H. Mössenböck. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Companion Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 29–32, 2019. DOI: <https://www.doi.org/10.1145/3302541.3313100>.
- [326] M. Weninger, L. Makor, and H. Mössenböck. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Proceedings of the Working Conference on Software Visualization (VISSOFT) 2020*, pages 110–121, 2020. DOI: <https://www.doi.org/10.1109/VISSOFT51673.2020.00017>.
- [327] M. Weninger, L. Makor, and H. Mössenböck. Memory Leak Visualization using Evolving Software Cities. In *Proceedings of the 10th Symposium on Software Performance (SSP)*, pages 44–46, 2019. URL: http://pi.informatik.uni-siegen.de/stt/39_4/01_Fachgruppenberichte/SSP2019/SSP2019_Weninger.pdf.
- [328] M. Weninger, L. Makor, and H. Mössenböck. Heap Evolution Analysis Using Tree Visualizations. In *Proceedings of the 11th Symposium on Software Performance (SSP)*, 2020. URL: https://www.performance-symposium.org/fileadmin/user_upload/palladio-conference/2020/Papers/SSP2020_paper_6.pdf.
- [329] M. Weninger, L. Makor, and H. Mössenböck. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor - Artifact (Binaries, Data Sets, Video, Instructions), 2020. DOI: <https://www.doi.org/10.5281/zenodo.3991785>.
- [330] M. Weninger, L. Makor, and H. Mössenböck. Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations. In *Proceedings of the Conference on Smart Tools and Applications in Graphics (STAG) - Eurographics Italian Chapter Conference*, 2020. DOI: <https://www.doi.org/10.2312/stag.20201241>.
- [331] M. Weninger and H. Mössenböck. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 115–126, 2018. DOI: <https://www.doi.org/10.1145/3184407.3184412>.
- [332] A. Wessels, M. Purvis, J. Jackson, and S. S. Rahman. Remote Data Visualization through WebSockets. In *Proceedings of the 8th International Conference on Information Technology: New Generations (ITNG)*, pages 1050–1051, 2011. DOI: <https://www.doi.org/10.1109/ITNG.2011.182>.
- [333] R. Wettel. Visual Exploration of Large-Scale Evolving Software. In *Companion Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 391–394, 2009. DOI: <https://www.doi.org/10.1109/ICSE-COMPANION.2009.5071029>.
- [334] R. Wettel. CodeCity, 2020. URL: <https://wettel.github.io/codecity.html>. last visited on 2021-04-08.
- [335] R. Wettel and M. Lanza. Program Comprehension Through Software Habitability. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*, pages 231–240, 2007. DOI: <https://www.doi.org/10.1109/ICPC.2007.30>.
- [336] R. Wettel and M. Lanza. Visualizing Software Systems as Cities. In *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 92–99, 2007. DOI: <https://www.doi.org/10.1109/VISSOFT.2007.4290706>.
- [337] R. Wettel and M. Lanza. CodeCity: 3D Visualization of Large-Scale Software. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 921–922, 2008. DOI: <https://www.doi.org/10.1145/1370175.1370188>.
- [338] R. Wettel and M. Lanza. Visual Exploration of Large-Scale System Evolution. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, pages 219–228, 2008. DOI: <https://www.doi.org/10.1109/WCRE.2008.55>.
- [339] R. Wettel, M. Lanza, and R. Robbes. Software Systems as Cities: A Controlled Experiment. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 551–560, 2011. DOI: <https://www.doi.org/10.1145/1985793.1985868>.
- [340] N. P. Wilde. Using Cognitive Dimensions in the Classroom as a Discussion Tool for Visual Language Design. In *Conference on Human Factors in Computing Systems: Common Ground (CHI)*, pages 187–188, 1996. DOI: <https://www.doi.org/10.1145/257089.257252>.
- [341] R. Williams. *The Animator’s Survival Kit—Revised Edition: A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Internet Animators*. Faber & Faber Inc., 2009.
- [342] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4), Jan. 2013. DOI: <https://www.doi.org/10.1145/2400682.2400689>.

- [343] U. Wolz, G. Carmichael, and C. Dunne. Learning to Code in the Unity 3D Development Platform. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE)*, page 1387, 2020. DOI: <https://www.doi.org/10.1145/3328778.3367010>.
- [344] A. Wöß, M. Löberbauer, and H. Mössenböck. LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. In *Proceedings of the Joint Modular Languages Conference on Modular Programming Languages (JMLC)*, volume 2789, pages 192–201, 2003. DOI: https://www.doi.org/10.1007/978-3-540-45213-3_25.
- [345] J. Wu and Y. Yuan. Improving Searching and Reading Performance: The Effect of Highlighting and Text Color Coding. *Inf. Manag.*, 40(7):617–637, 2003. DOI: [https://www.doi.org/10.1016/S0378-7206\(02\)00091-5](https://www.doi.org/10.1016/S0378-7206(02)00091-5).
- [346] G. H. Xu. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs. In A. L. Hosking, P. T. Eugster, and C. V. Lopes, editors, *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 111–130, 2013. DOI: <https://www.doi.org/10.1145/2509136.2509512>.
- [347] G. H. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011. DOI: <https://www.doi.org/10.1145/1993498.1993530>.
- [348] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing. *ACM Trans. Softw. Eng. Methodol.*, 23(3):23:1–23:50, 2014. DOI: <https://www.doi.org/10.1145/2560047>.
- [349] G. H. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-scale Object-oriented Applications. In *Proceedings of the Workshop on Future of Software Engineering Research (FoSER)*, pages 421–426, 2010. DOI: <https://www.doi.org/10.1145/1882362.1882448>.
- [350] G. H. Xu and A. Rountev. Precise Memory Leak Detection for Java Software using Container Profiling. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 151–160, 2008. DOI: <https://www.doi.org/10.1145/1368088.1368110>.
- [351] G. H. Xu and A. Rountev. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010. DOI: <https://www.doi.org/10.1145/1806596.1806616>.
- [352] G. H. Xu and A. Rountev. Precise Memory Leak Detection for Java Software Using Container Profiling. *ACM Trans. Softw. Eng. Methodol.*, 22(3):17:1–17:28, 2013. DOI: <https://www.doi.org/10.1145/2491509.2491511>.
- [353] D. Yan, G. H. Xu, S. Yang, and A. Rountev. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *Proceedings of the 12th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, page 87. ACM, 2014. URL: <https://dl.acm.org/citation.cfm?id=2544151>.
- [354] YourKit. YourKit Java Profiler, 2021. URL: <https://www.yourkit.com>. last visited on 2021-06-15.
- [355] H. Yu, X. Shi, and W. Feng. LeakTracer: Tracing Leaks Along The Way. In M. W. Godfrey, D. Lo, and F. Khomh, editors, *Proceedings of the 15th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 181–190, 2015. DOI: <https://www.doi.org/10.1109/SCAM.2015.7335414>.
- [356] Q. Yu, S. Jiang, and Y. Liu. A Detection and Measurement Approach for Memory Leaked Objects in Java Programs. *IEICE Trans. Inf. Syst.*, 98-D(5):1053–1061, 2015. DOI: <https://www.doi.org/10.1587/transinf.2014EDP7320>.
- [357] S. Zaman, B. Adams, and A. E. Hassan. A Large Scale Empirical Study on User-Centric Performance Analysis. In *Proceedings of the Fifth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 410–419, 2012. DOI: <https://www.doi.org/10.1109/ICST.2012.121>.
- [358] N. Zhang, N. Jiang, Y. Zhang, and G. Huang. Towards Automated Generation of User-Specific Eclipse Wizard. In *Proceedings of the International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*, pages 490–497, 2010. DOI: <https://www.doi.org/10.1109/CyberC.2010.95>.
- [359] H. Zhao and L. Lu. Variational Circular Treemaps for Interactive Visualization of Hierarchical Data. In *Proceedings of the IEEE Pacific Visualization Symposium (PacificVis)*, pages 81–85, 2015. DOI: <https://www.doi.org/10.1109/PACIFICVIS.2015.7156360>.
- [360] T. Zimmermann and A. Zeller. Visualizing Memory Graphs. In *Software Visualization*, pages 191–204, 2001. DOI: https://www.doi.org/10.1007/3-540-45875-1_15.

	<h2 style="margin: 0;">Curriculum Vitae</h2> <h1 style="margin: 0;">Dipl.-Ing. Markus Weninger</h1> <p style="margin: 0;"> 📍 4113 St. Martin im Mühlkreis, Schulstraße 24 ✉ markus.weninger@jku.at ☎ +43660/3115418 </p>
Birth Date:	04.04.1992, Vöcklabruck, Austria
Education:	<p>Since 2017: Doctorate degree – Computer Science Johannes Kepler University Linz Thesis: Detection and Analysis of Memory Anomalies in Managed Languages Using Trace-Based Memory Monitoring</p> <p>2015 - 2017: Master - Computer Science / Software Engineering (with distinction) Johannes Kepler University Linz Thesis: User-defined Classification and Multi-grouping of Data in a Memory Monitoring Tool</p> <p>2012 - 2015: Bachelor - Informatik (with distinction) Johannes Kepler University Linz Thesis: An Experiment to Measure the Performance Trade-off between Traditional IO and Memory-mapped Files</p> <p>2006 - 2011: Higher education (with distinction) HTL Leonding, EDV und Organisation (Upper Secondary Technical and Vocational College – Department for Informatics) Thesis: Digital Teaching System – Digitales Lehrsystem an der HTL Leonding</p>
Teaching Experience:	<p>Grundlagen der Programmierung Basics of Software Development 2017 WS - German</p> <p>Softwareentwicklung 1 Software Development 1 2020 WS – German</p> <p>Softwareentwicklung 2 Software Development 2 2018 SS, 2019 SS, 2020 SS, 2021 SS – German</p> <p>Übersetzerbau Compiler Construction 2018 WS, 2019 WS, 2020 WS – German / English</p> <p>Thesis supervision: 10 bachelor theses supervised, 6 master theses co-supervised, 5 master software projects (7.5 ECTS) supervised</p> <p><i>WS = Winter Semester, SS = Summer Semester all at Johannes Kepler University Linz</i></p>
Professional Experience:	<p>Since 09/2017: Institute Assistant / Researcher Johannes Kepler University, Institute for System Software – Linz</p> <p>09/2017 – 01/2020: Researcher Johannes Kepler University, Christian Doppler Laboratory MEVSS - Linz</p> <p>03/2017 – 07/2017: Tutor "Software Processes and Tools" Johannes Kepler University, Institute for Software Systems Engineering – Linz</p> <p>10/2016 – 02/2017: Tutor "Requirements Engineering" Johannes Kepler University, Institute for Software Systems Engineering – Linz</p> <p>10/2015 – 08/2017: Student Researcher Johannes Kepler University, Christian Doppler Laboratory MEVSS – Linz</p> <p>05/2013 – 09/2015: Software Engineer C#/.Net bet-at-home.com Entertainment GmbH – Linz</p> <p>07/2010 – 08/2010: Intern - Software Engineer MIC Customs Solutions / MIC Datenverarbeitung GmbH – Linz</p> <p>07/2008- 08/2008: Intern – IT AIM Technical Solutions GmbH – Timelkam</p>

Languages:	German (native) English (fluent)
Workshops:	<p>Events:</p> <ul style="list-style-type: none"> CoderDojo Frauen in die Technik JKU Science Holidays JKU Workshops Sekundarstufe II KinderUni Tomorrow's Experts in Computing Traumberuf Technik Young Computer Scientists ... and others ... <p>Topics:</p> <ul style="list-style-type: none"> JavaFX Game Programming 3D Game Programming Kreatives Programmieren mit micro:bit Creative Programming with micro:bit Spieleentwicklung mit Scratch Game Programming with Scratch Spielegerisches Kennenlernen von Sortieralgorithmen Sorting is Fun ... and others ...
Volunteer Work:	<p>Students Union "Informatik & AI" Johannes Kepler University Linz</p> <p>Students Union "PhD Studies - Engineering & Natural Sciences" Johannes Kepler University Linz</p> <p>Rotaract Social service and community service, Linz</p>
Awards:	<p>Best Paper "Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations", Conference on Smart Tools and Applications in Graphics (STAG), 2020</p> <p>Best Paper "Memory Cities: Visualizing Heap Memory Evolution Using The Software City Metaphor", IEEE Working Conference on Software Visualization (VISSOFT), 2020</p> <p>Best Presentation of Best Paper Candidates "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring", International Conference on Performance Engineering (ICPE), 2018</p> <p>Best Paper Candidate "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring", International Conference on Performance Engineering (ICPE), 2018</p> <p>2nd place of the Adolf-Adam Price (price for the best informatics master thesis) Johannes Kepler University, 2017</p> <p>Winner of the Coding 4 a Cause (C4C:EU) Contest Association for the Advancement of Assistive Technology in Europe (AAATE) conference, 2015.</p> <p>Multiple "Top 10%" results at Catalysts Coding Contests / Cloudflight Coding Contests Linz / Vienna / Online, 2013 – 2020</p>



Publications

Dipl.-Ing. Markus Weninger

©4113 St. Martin im Mühlkreis, Schulstraße 24 ✉markus.weninger@jku.at 📞+43660/3115418

Publications:

- [18] *Weninger, M.; Gander, E.; Mössenböck, H.*,
"Guided Exploration: A Method for Guiding Novice Users in Interactive Memory Monitoring Tools", ACM HCI (EICS) 2021
- [17] *Weninger, M.; Makor, L.; Mössenböck, H.*,
"Memory Leak Analysis using Time-Travel-based and Timeline-based Tree Evolution Visualizations", STAG 2020 (Best Paper)
- [16] *Weninger, M.; Makor, L.; Mössenböck, H.*,
"Heap Evolution Analysis Using Tree Visualizations", SSP 2020
- [15] *Weninger, M.; Gander, E.; Mössenböck, H.*,
"Investigating High Memory Churn via Object Lifetime Analysis to Improve Software Performance", SSP 2020
- [14] *Weninger, M.; Makor, L.; Mössenböck, H.*,
"Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor", VISSOFT 2020 (Best Paper)
- [13] *Weninger, M.; Grünbacher, P.; Gander, E.; Schörghumer, A.*,
"Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study", ACM HCI (EICS) 2020
- [12] *Weninger, M.; Makor, L.; Mössenböck, H.*,
"Memory Leak Visualization using Evolving Software Cities", SSP 2019
- [11] *Weninger, M.; Gander, E.; Mössenböck, H.*,
"Detection of Suspicious Time Windows in Memory Monitoring", MPLR 2019
- [10] *Weninger, M.; Makor, L.; Mössenböck, H.*,
"AntTracks TrendViz: Configurable Heap Memory Visualization Over Time", ICPE 2019
- [9] *Weninger, M.; Gander, E.; Mössenböck, H.*,
"Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection", ICPE 2019
- [8] *Weninger, M.; Grünbacher, P.; Zhang, H.; Yue, T.; Ali, S.*,
"Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment", APSEC 2018
- [7] *Weninger, M.; Gander, E.; Mössenböck, H.*,
"Analyzing the Evolution of Data Structures in Trace-Based Memory Monitoring", SSP 2018
- [6] *Weninger, M.; Gander, E.; Mössenböck, H.*,
"Utilizing Object Reference Graphs and Garbage Collection Roots to Detect Memory Leaks in Offline Memory Monitoring", ManLang 2018
- [5] *Weninger, M.; Mössenböck, H.*,
"User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring", ICPE 2018 (Best Paper Candidate)

	<p>[4] <i>Weninger, M.; Lengauer, P.; Mössenböck, H.</i>, "User-centered Offline Analysis of Memory Monitoring Data", ICPE 2017</p> <p>[3] <i>Lengauer, P.; Bitto, V.; Mössenböck, H.; Weninger, M.</i>, "A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008", ICPE 2017</p> <p>[2] <i>Lengauer, P.; Bitto, V.; Fitzek, S.; Weninger, M.; Mössenböck, H.</i>, "Efficient Memory Traces with Full Pointer Information", PPPJ 2016</p> <p>[1] <i>Weninger, M.; Ortner, G.; Hahn, T.; Drümmer, O.; Miesenberger, K.</i>, „ASVG - Accessible Scalable Vector Graphics: intention trees to make charts more accessible and usable“, Journal of Assistive Technologies, Vol. 9 Issue 4, 2015</p>
--	---