JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Markus Hirth, BSc

Submitted at
Institute for System Software

Supervisor
Dipl.-Ing. Dr. Markus Weninger, BSc

December 2023

Analyzing Collection Staleness and Collection Anti-Patterns that Lead to Memory Problems

Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Dipl.-Ing. Dr. Markus Weninger, Bsc
Institute for System Software
T +43-732-2468-4361
markus.weninger@jku.at

Master's Thesis
Analyzing Collection Staleness and Collection Anti-Patterns that Lead to Memory Problems

Student: Markus Hirth
Advisor: Dipl.-Ing. Dr. Markus Weninger, BSc
Start date: October 2022

Modern programming languages such as Java are so-called *managed languages* that leverage garbage collection to free the programmer from the error-prone task of manually managing memory. While garbage collection prevents certain memory problems, anomalies such as memory leaks can still happen. Such anomalies are often connected to the inproper use of collections such as lists or maps.
For example, Xu and Rountev presented “Precise Memory Leak Detection for Java Software Using Container Profiling” (https://doi.org/10.1145/2491509.2491511, https://doi.org/10.1145/1368088.1368110) in which they present a technique on how to track accesses to containers, how to calculate the staleness of the container’s elements (i.e., when each element was accessed the last time) and how this information, together with information about the collection’s overall size and growth, can be used to calculate a leaking confidence factor. This leaking confidence factor tells how probable it is that a collection exhibits a memory growth problem.

In previous work (“Collecting Memory Monitoring Data using Aspect-oriented Programming” by Markus Hirth), we have shown that it is possible to collect information about collections (creation, additions, removes, accesses, size) in Java using aspect-oriented programming, opposed to the established way of using (native) agents and bytecode manipulation. This information is written to a trace file while the monitored application is running.

The goal of this master’s thesis is to utilize such trace files to analyze the staleness of collections and to inspect general anti-patterns that can lead to memory problems. As a first step, the student has to develop a parser that can process the trace files. Secondly, inspired by the work by Xu and Rountev, the student should develop data structures that allow us to reconstruct staleness information similar to the one presented in their papers – this should allow us to also calculate a leaking confidence factor. The student might also have to improve the trace collection with new event types and information for this. Once the subgoal of replicating the leaking confidence factor metric is achieved, the trace collection, the parser and the data structures should be extended to detect other memory anti-patterns. The student might find inspiration in “Patterns of Memory Inefficiency” by Chis et. al (https://doi.org/10.1007/978-3-642-22655-7_18) regarding possible patterns that might be detected using the trace data.
The written thesis should also contain an evaluation of the implemented approach, showing how the resulting tool can be used to detect memory problems in applications and how the tool helps users fixing these problems.

Modalities:
The progress of the project should be discussed at least every four weeks with the advisor. A time schedule and a milestone plan must be set up within the first 3 weeks and discussed with the advisor and the supervisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 30.09.2023.

Abstract

Executing programs allocates memory on the system. As a result, it is vital, that programs use the memory efficiently. But bugs in the program code can lead to unnecessary consumed memory, a memory leak. Because of this, it is essential to identify possible memory leaks in the program. Hence, there are different approaches with their own advantages and disadvantages to identify possible memory leaks. With our approach, we want to achieve similar results to an existing approach and want to explore whether there are advantages and disadvantages, when tackling the problem of memory leak analysis with the concept of aspect-oriented programming (AOP). Furthermore, we explored if it is possible to identify patterns of memory inefficiency in the data of the tracked program.

Kurzfassung

Die Ausführung von Programmen allokiert Speicher auf dem System. Daraus resultiert, dass es für Programme unabdingbar ist, den Speicher effizient zu nutzen. Aber Fehler im Programmcode können zu unnötig genutzten Speicher führen, genannten Speicherlecks. Dadurch ist es unerlässlich, mögliche Speicherlecks in Programmen zu identifizieren. Deshalb existieren viele verschiedene Ansätze, wobei jeder Ansatz seine eigenen Vorteile und Nachteile besitzt, um mögliche Speicherlecks zu identifizieren. Wir wollen mit unserem Ansatz ähnliche Resultate zu einem bereits existierenden Ansatz erreichen und erforschen dabei, ob es Vorteile und Nachteile gibt, wenn wir das Problem mit dem Konzept der aspekt-orientierten Programmierung (AOP) angehen. Außerdem erforschen wir, ob es möglich ist, bereits vordefinierte Muster für Speicherineffizienz in den Daten des nachverfolgten Programmes zu identifizieren.

Table of Content

Contents

1 Introduction

A lot of variables and data structures are used over the execution time of a program. The program needs a space for the values assigned to these variables and data structures. This space is provided by the system on which the program runs. As there is only a limited amount of memory in a system, the programs running on the system share the memory. If a single program consumes more memory than it should over its execution time, there is a high chance that a variable or a data structure in the program is unnecessarily kept alive. This phenomenon is known as leaking memory. This causes a problem, because if a lot of programs have leaking memory problems, the limited memory of the system will be used up far faster and fewer programs can request memory from it.

There are different possibilities how it comes to a leaking memory problem. For example, if the program developer needs to manage the memory manually, it could be that he forgot to free the memory. Another example would be that elements are added to data structures, even though they are never accessed or removed afterwards. This way, these elements consume memory without being used until the data structure is deallocated. There are different approaches to identify memory leaking problems or prevent the memory leaking problems. When manual memory management in programming languages (for example C and C++) causes problems, then the problem can be prevented by designing programming languages where memory management is automated (for example Java). But such a prevention mostly has advantages and disadvantages. On the one hand, the automatic memory management solves the problem of forgetting to free memory. On the other hand, to automate the memory management comes with a program execution time overhead. When prevention for a type of memory leak is not possible or the disadvantages of the prevention outweigh the advantages, then the memory leak needs to be identified on the source code level. There are different approaches, some of which can only be applied on certain programming languages. [1]

One such approach was described by Xu et al. [30]. In the programming language Java, they modified selected collection classes to collect which operations are performed on them. These operations were classified into three types: adding, getting and removing. The memory consumption was monitored through the use of Java agents. The gathered data was then used to calculate two metrics, the *staleness*, which is generally defined as the unused time between the removal and the last access, and the *memory consumption*, which were then combined into a so-called *leaking confidence* score. This leaking confidence score shows how likely it is that a collection leaks memory.

Our approach in this work uses different concepts to try to achieve similar results. For gathering the operations on the collections, the concept of *aspect-oriented programming* (AOP) is used. AOP enables us to extract crosscutting concerns, such as gathering data before and/or after methods with a certain name pattern. As we tried to not use Java agents, gathering data on memory consumption of the collections became extremely difficult. In the end, we decided to not collect information about the memory consumption, as the scalability for more collections or larger collections performed extremely poorly. Once we finalized the implementation of gathering staleness data, a parser was written to process the data, so that it can be visualized to the user in the frontend. In the frontend, the user can interact with the visualized data, to find bad code practices, that are defined as anti-patterns for memory inefficiency.

The following chapter will first introduce terminologies that are used throughout the work.

Then, a short overview of our approach is provided, where the connections between the steps of gathering the data to visualizing the gathered data are shortly explained. Afterwards, each part of the approach is explained in more detail. After explaining the approach, the results of applying our approach on a known memory leaking application and how well the approach performed for this problem are presented. Afterwards, limitations that were encountered, such as the scalability of memory consumption analysis using AOP, will be shortly discussed, before presenting possible ways on how to improve and extend the work in the future.

2 Background

This chapter introduces concepts, that are needed for understanding parts of this work.

2.1 Java

Java is a high-level programming language with automatic memory management, that the applications we test in this work are written in. For that reason, we want to clarify certain Java-specific terminologies that are will be encountered throughout this work.

Java Virtual Machine The Java Virtual Machine (JVM) is an abstract computing machine, that has its own instruction set (Java bytecode, also called class files) and manipulates various memory areas at run time. As long as a programming language can compile to Java's class file format, the JVM is able to host it. Because of this, the JVM knows nothing of the Java programming language per-se, but it can host the binary format that Java applications are compiled to. [22]

Collection Types Java has a collection framework that is based on two main interfaces, `Collection` and `Map`, which other collections inherit from. Most collection types are based on the `Collection` interface and extend it with their own specific functionalities. `Map` is the basis for the collection types that map keys to values. Each of the two main interfaces provides a specific set of basic method calls on which the collection framework is build upon. [15, 19]

In this work the following collection types (that implement one of the two main interfaces) are directly mentioned:

- *`HashMap`*: `HashMap` is a hash-table-based implementation of the `Map` interface. It allows `null` values and is not suitable for the access of multiple threads at the same time, meaning that it is not thread safe. [17]
- *`ConcurrentHashMap`*: A `ConcurrentHashMap` is also based on a hash table such as the `HashMap`. But the `ConcurrentHashMap` is a thread-safe variant. [16]
- *`HashSet`*: A `HashSet` is backed by a hash table and implements the `Set` interface, which inherits from the `Collection` interface. Because of this, a `HashSet` allows no duplicate elements. The `HashSet` is not a thread-safe collection type. [18]
- *`ArrayList`*: An `ArrayList` is a resizable-array-based implementation of the `List` interface, which inherits from the `Collection` interface. It expands the functionality by methods for manipulating the size of the internal array. An `ArrayList` is not thread-safe. [14]

System.identityHashCode The method `System.identityHashCode(Object x)` returns a hash code for objects that will be calculated based on the original implementation for hash code in the `Object` class. This means that, even if a class has overridden the hash-Code method, the `Object` hash-Code method will be used. [21]

WeakReference A `WeakReference` references a object weakly. When a garbage collector determines that an object is only reachable through a weak reference, the garbage collector will clear the object and free the memory. [23, 25]

2.2 Memory Management

Programming languages need memory to save values and information. For this reason, memory management that controls and coordinates the computer memory is essential. Programming languages such as C or C++ use manual memory management, where the memory has to be allocated and deallocated by the developer. But it is well known that manual memory management can cause many problems. An example is to forget the deallocations of objects that are no longer needed, resulting in wasted heap memory. Opposed to manual memory management is the automatic memory management, that is used in many high-level object-oriented programming languages (OOP) such as Java. The automatic memory management uses efficient garbage collection algorithms to free the memory from objects that are no longer referenced from the program. The garbage collection operations are performed automatically at regular intervals. Consequently, automatic garbage collectors (GC) eliminate many difficulties associated with manual memory management such as forgotten deallocations and wrongly freed memory at the cost of performance. [1]

As the approach presented in this work focuses on Java, the memory management of the JVM has to be considered. The JVM has three core components for efficient memory management that developers need to be aware of, the heap, the compiler and the garbage collector [1].

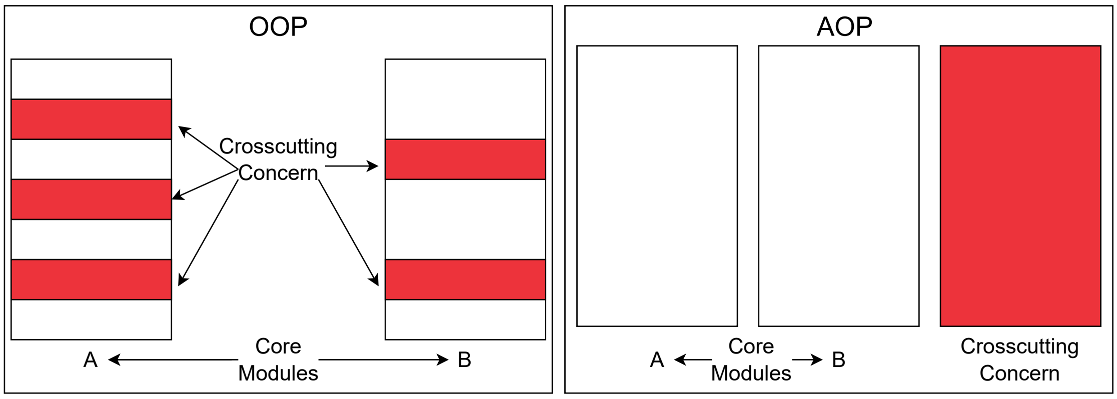
Heap memory The heap memory holds the objects used by a program. The JVM forms the heap memory at startup and all object instances produced at run time share it. The JVM is also responsible for initiating automatic memory management operations for the heap memory. [1]

Just-In-Time (JIT) Compiler Just-In-Time compilation controls the operating program by tracing data to make decisions with the goal of memory optimizations. The JIT compiler repeatedly analyzes the executed code for areas where the cost of compiling is outweighed by the benefits of the optimizations. This makes the JIT compiler a form of dynamic compilation that enables adaptive optimization. [1]

Garbage Collector (GC) The garbage collector removes unreferenced objects from the heap memory. This frees memory space to be able to create new objects. In the JVM, a GC automatically takes care of the memory management. This frees the developer from manually managing the memory and thus prevents its possible mistakes. However, this error prevention comes with the cost of extra run-time overhead. [1]

2.3 Aspect-oriented Programming

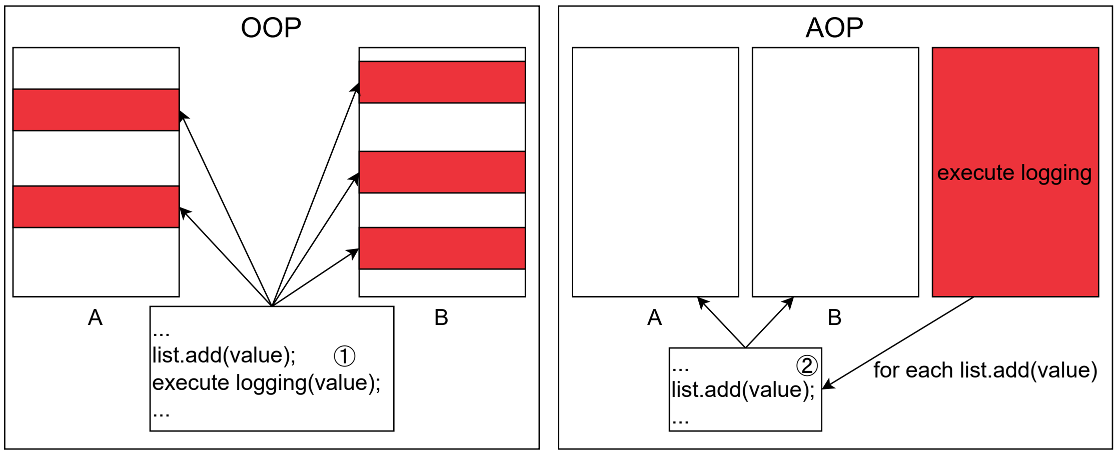
Aspect-oriented programming (AOP) is a methodology that separates crosscutting concerns from core concerns. The core concerns are the primary functionality of the program that are bundled in core modules. Crosscutting concerns, such as logging, are scattered across multiple core modules, as shown on the left (a) in Figure 1. This leads to the undesirable situation, where the introduction of new crosscutting concerns or the modification of existing crosscutting concerns requires the modification of all the relevant core modules. Shown on the right (b) in Figure 1 is the separation of the crosscutting concern from the core modules. To achieve separation of core concerns and crosscutting concerns, AOP introduces a new unit of modularization, Aspects, which crosscuts the other modules. For a better manageability it is recommended to have one crosscutting concern per Aspect. Aspects also need to be combined with the core modules. This combination of the core modules and Aspects is done by the Aspect weaver, a compiler-like entity. Consequently, the process of combination is called weaving. When a new crosscutting concern is added, only a new Aspect has to be created. Similarly, to modify existing crosscutting concerns, only the existing Aspect must be modified. [10]



(a) The coupling of core concerns and crosscutting concerns in different core modules practiced in OOP.

(b) The separation of the crosscutting concern from the core modules into an aspect.

Figure 1: The different approaches to crosscutting concerns from OOP and AOP.



(a) Logging values that are added to lists in OOP.

(b) Logging values that are added to lists in AOP.

Figure 2: OOP and AOP have the same behavior when executed.

Example In Figure 2 we see an example of writting a log message whenever an element is added to a list, once in the OOP-methodology on the left (a) and once in the AOP-methodology on the right (b). Both methodologies result in the same runtime behavior. We see in (a) that at least one call to the logging method ① needs to be done every time an add to the list occurs. When additional information should be logged later on, each call to the logging method needs to be changed in each core module. In contrast, we see in (b) that the execution of the logging code was separated into an aspect. The aspect catches when a list adds an element and executes the logging code. Therefore, we just need to modify the aspect later on, when additional information should be logged.

2.3.1 AspectJ

One concrete implementation of AOP for Java is called *AspectJ*. It has a compiler named *ajc* that allows to weave the AspectJ code directly into a Java application. Weaving can be done while compiling the Java application or later in the already compiled Java binaries. When weaving while compiling, it is called compile-time weaving. Weaving Java binaries after compilation is called post-compile time weaving. For the weaving to be possible, ajc is able to also compile Java files. [10]

The AspectJ code is defined by four main terminologies, as described in [10], which will be demonstrated in the form of an example after giving their definitions:

1. *Join point*: A join point is an identifiable point of execution in the program. Some examples for such identifiable execution points are method calls and assignments to variables.
2. *Pointcut*: The pointcut specifies which join points should be captured in the program execution. They can be declared in aspects and Java classes.
3. *Advice*: Advices specify which actions should be executed at caught join points specified by a pointcut. The actions can be executed in three different ways: (1) Before the join point is executed, (2) after the join point is executed or (3) around the join point. The around execution of a join point is a little special, because the caught join point must be executed from within the advice. Also, when a value is returned from a joined method, that return value must also be returned manually from within the advice.
4. *Aspect*: An Aspect contains mainly definitions of pointcuts and advices. Therefore, aspects are constructs similar to Java classes. Yet, they cannot be instantiated directly. Also, Aspects can only inherit from abstract aspects and from Java classes.

Limitations AspectJ cannot weave the Java standard library without especially modifying the Java project through special methods, such as Java bytecode instrumentation. But this special methods bring their own risk. For example, Java bytecode instrumentation is difficult and may crash the JVM. [26]

Example In this short example we examine a small Aspect shown in Listing 1. In the first line ①, we see that similar to a Java class, there is an access specification, the keyword `aspect` instead of the keyword `class` and the name. The second line ② defines a pointcut that catches all calls to add methods (independent of their parameters, thus the .. in the parameter list) of

`List` (and inheriting classes, thus the + after `List`) objects. The * indicates that the access modifier does not matter. Then, we create an advice ③ that executes code before each join point caught by the specified pointcut. We also can define another advice ④ that executes code after the same join points. As we can see, one advice is locked to one execution way, but we are still allowed to simply create another advice.

```
public aspect ListAddAspect { ①
    pointcut callAllListAdd() : call(* List+.add(..)); ②

    before () : callAllListAdd() { ③
        System.out.println("Before adding element!");
    }

    after () : callAllListAdd() { ④
        System.out.println("After adding element!");
    }
}
```

Listing 1: Example AspectJ Aspect.

2.4 Javascript Library d3

The d3 library (which stands for data-driven-documents) is a free, open-source JavaScript library that takes a low-level approach for visualizing data. The library was created by Mike Bostock in 2011 and has since then been used for data visualizations in newsrooms, websites and personal portfolios. Because of its low-level approach, d3 does not provide ready-to-use charts. Instead, it allows the flexible design of charts as needed, providing easy-to-use utility functions. Thus, it is often used in higher-level charting libraries as the foundational building block. [12]

Example - Rectangle To understand how d3 works, a small example provides the best way to explain it. Listing 2 shows a small HTML file. First, we see the declaration that it is an HTML file. Then we declare the head, where we define the title of the page, and more importantly, load in a script tag the d3 library ①. Following the header is the body, where we define an svg tag ②. The svg tag provides us with a space where we can draw on. Now that the d3 library is loaded and a space to draw on was created, we start our script. First, we select ③ the drawing space through d3 and save the selection in a variable. Then we define an array with the information (the starting point defined by x and y, the width and height of the rectangle and the color of the rectangle) of a single green rectangle and an array with the information of two different rectangles ④. Afterwards, we define a redraw function ⑤ that takes an array and draws it onto the drawing space. To achieve this, in the redraw function all existing rectangles in the drawing space, which we saved prior to this in a variable, are selected. After setting the rectangle data from the parameters for the selection, the join appends or updates a rectangle through the provided data and deletes rectangles that are no longer present in the data. Then the position, size and color of each rectangle is set through the information provided by the data. Now we call the redraw function two times, first with an array of an green rectangle and then with an array of an red rectangle and blue rectangle ⑥. This results in first drawing a green rectangle which is shown in Figure 3. Then the second redraw draws a red rectangle and blue rectangle and deletes the green rectangle because it does not exist in the array. Figure 4 shows the result of the second redraw. In this example, the second redraw should happen so fast, that the green rectangle should not be noticed at all. Finally, we draw a border at the edges of the drawing space ⑦ for a better visibility in Figure 3 and Figure 4.

This short example should help to understand that using d3 data can be transformed into visualization quite easily. For example, developing a bar chart is achieved quite easily, while still having the possibility to change little details, which would not be possible with other libraries.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <script type="text/javascript"
       src="https://unpkg.com/d3@7.8.4/dist/d3.min.js"> ①
    </script>
    <title>Rectangle Example</title>
</head>
<body>
    <svg id="rectangleSVG" width="300" height="300"></svg> ②

    <script type="text/javascript">
        let rectSVG = d3.select("#rectangleSVG"); ③
        let rectDelete =
            [{x: 50, y:50, w: 100, h: 100, c: "green"}];
        let rectsArray =
            [{x: 100, y: 100, w: 50, h: 100, c: "red"},
            {x: 200, y: 80, w: 20, h:200, c: "blue"}]; ④
        function redraw(rects) { ⑤
            rectSVG
                .selectAll("rect")
                .data(rects)
                .join("rect")
                .attr("transform", d =>
                   ‘translate(${d.x},${d.y})‘)
                .attr("width", d => d.w)
                .attr("height", d => d.h)
                .attr("fill", d => d.c);
        };
        redraw(rectDelete);
        redraw(rectsArray); ⑥
        rectSVG.append("rect") ⑦
            .attr("width", 300)
            .attr("height", 300)
            .attr("fill", "none")
            .attr("stroke", "black")
            .attr("stroke-width", 3);
    </script>
</body>
</html>
```

Listing 2: Example d3 code.

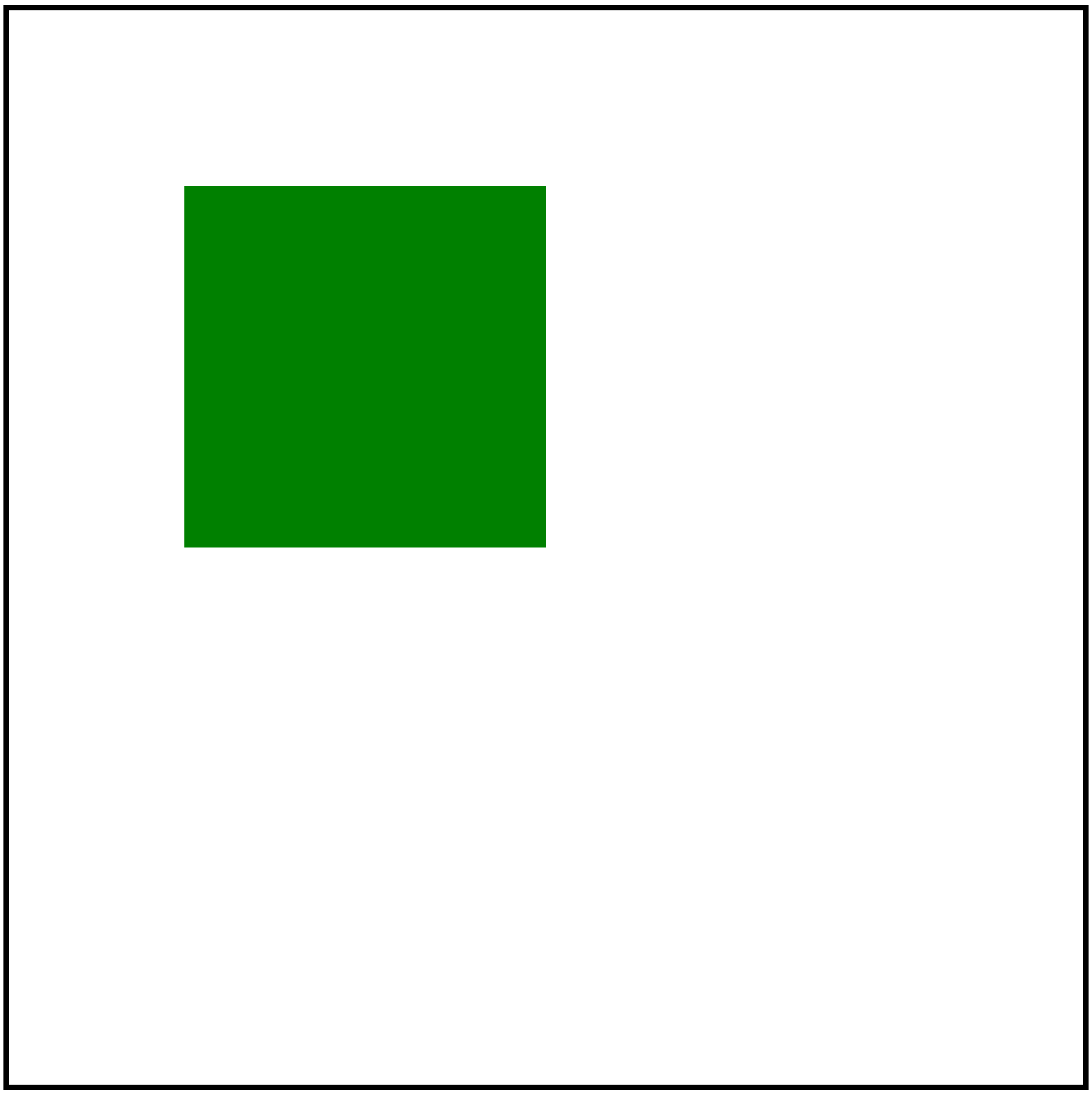


Figure 3: The deleted green rectangle of Listing 2.

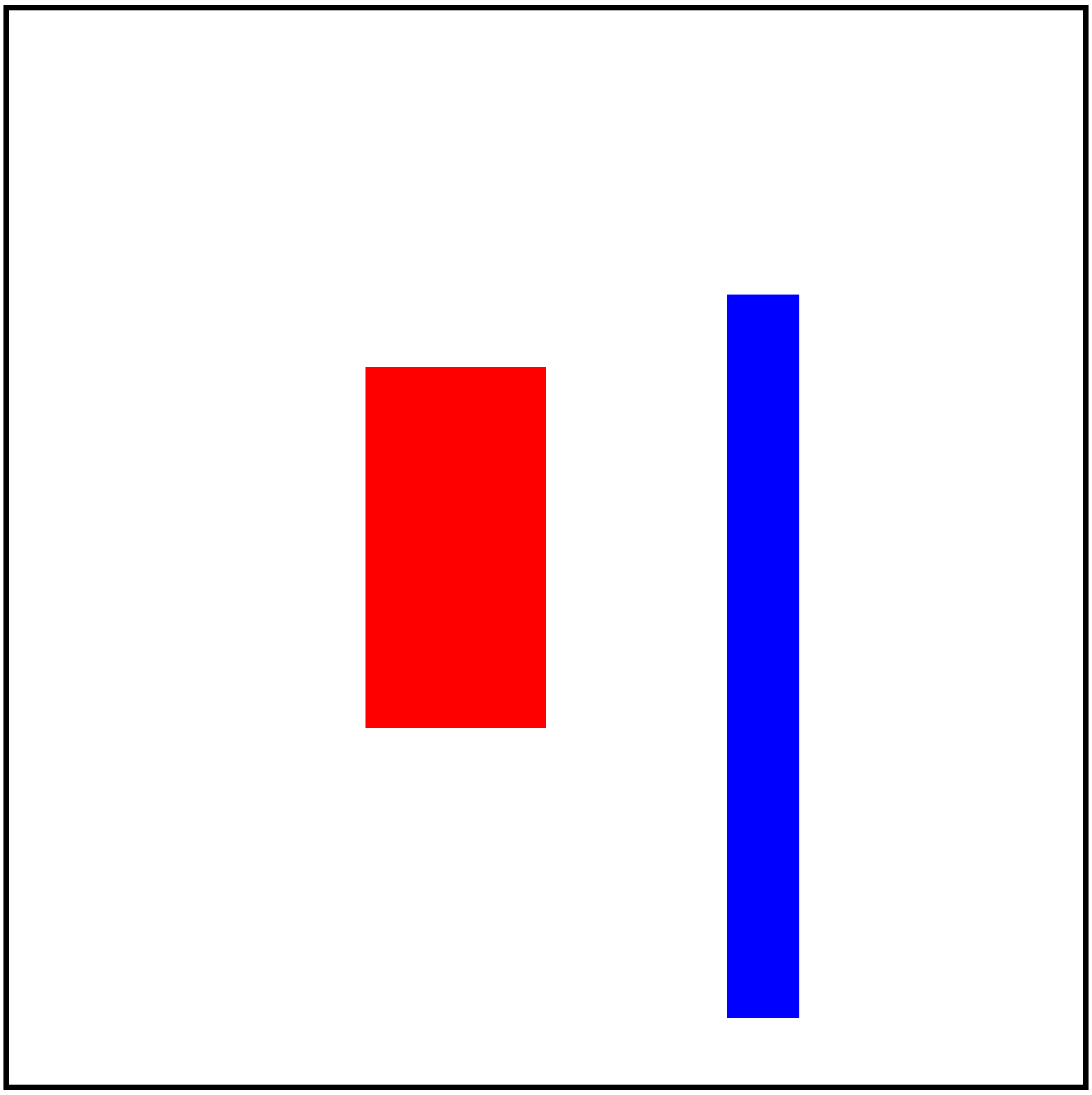


Figure 4: Resulting view of Listing 2.

3 Overview

Memory leaks occur because of different reasons in a program. In the programming language Java, the GC handles the allocation and deallocation of memory, which eliminates memory leaks through manual allocation and deallocation. Instead, when an object is referenced in the program, for example from a local or static variable, the GC is not able to free the memory. Furthermore, objects are often saved as elements in long-living collections and maps and might be forgotten to be removed or might be kept alive far longer than necessary. To track the program's execution to identify the leaking collections, Java offers various approaches. A well known concept is the concept of (native) java agents, that attach to the execution of a Java program and are able to access the JVM deeply to monitor the memory changes. In this work, we try to achieve similar results with a different approach based on the concept of *AOP* by modifying the code prior to execution. Then the gathered data is visualized to show possible memory leaks.

Steps of data gathering and analyzing

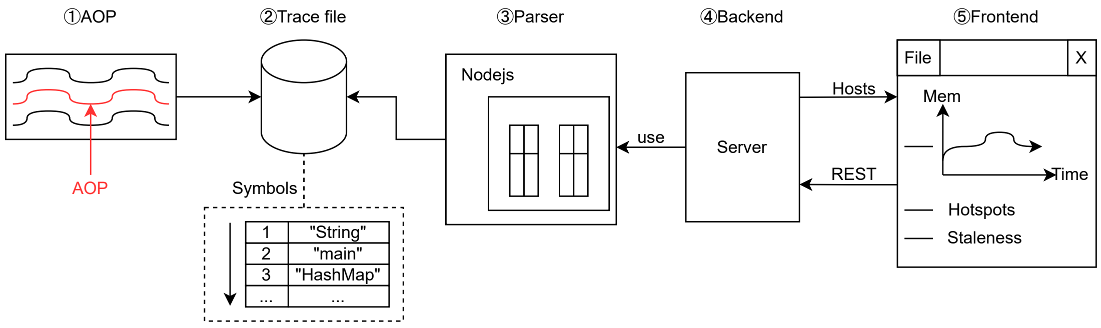


Figure 5: Overview of the steps of data gathering and analysis.

Figure 5 presents the steps that lead from gathering and saving the data with AOP, to the visualization in the frontend. The following list describes each of them shortly, while the explanation in more detail is done throughout the rest of the work.
① *AOP* is used to weave aspects and Java classes for all Java classes of the chosen application. The aspects catch calls to specific methods of Java classes. More specifically, all calls to methods of `Collection` and `Map` objects. Information about the respective method call is then saved to a trace file.
② The *trace file* contains events about five different kinds of method calls: allocation of collections, adds to collections, reads from collections, removes from collections and deallocations of collections. The allocation of collections contains the specific type of allocated `Collection`, the adding of collections contains the specific class name of the added element and all kinds except the deallocation contain at which source code location (file name + line number) the method call occurred. As this information is frequently the exact same and takes far more disk space to save, a separate *Symbols file* with the information is created. Only integer numbers that reference the content are saved in the trace file directly. That means a unique integer number was assigned to

the information and then both are saved in the Symbols file, where they can be looked up when needed.
③ The *parser* was written in Typescript and runs on the runtime environment *nodejs* [13]. The parser reads the trace file and the symbols file and reconstructs the call history for each `Collection` separately. Also, the calculation of staleness for each `Collection` and its elements for a specified time frame is done by the parser.
④ The *backend* hosts the frontend, creates when requested the parser and manages the communication between the frontend and the parser.
⑤ The visualization is done in the browser. Because of this, the visualization is separated from the parser into the frontend, which accesses the data through representational state transfer (*REST*) interface [7]. The visualization in the frontend enables the user to find *hotspots* in the program through choosing an interesting time frame. The chosen time frame is then more deeply analyzed on the *staleness* of the elements in the collections.

4 Trace File

After the brief introduction in Section 3, this chapter goes into more detail on how AOP is used to collect the data and how the trace file and symbols file are created from the collected data. Afterwards, the structure of the trace file and the structure of the symbols file are introduced in more detail.

4.1 Data Collection with AOP

This work uses AOP to weave Aspects and Java classes into already compiled Java classes and collect data on collections. This data is then steadily written into the trace file and symbols file.

Figure 6 shows the AspectJ aspects and Java classes and their interactions with each other. The `ConstructorAspect` ① catches all collection allocations (i.e., the creation of `Collection` objects and `Map` objects), defined in Table 2, through its defined pointcuts. Only the explicitly excluded Java classes or Aspects in the notWithin pointcut(which is comprised of all Aspects and Java classes in Figure 6) are ignored. The allocated `Collection` or `Map` is then saved in a `HashMap` in *ObjectIdentifier* ② with its `System.identityHashCode(Object x)` as key and a `WeakReference<Object>` as value. Every GC run, every entry in the `HashMap` is checked, because when its value is `null`, the collection was deallocated. The `CollectionAspect` ③ catches methods of the `Collection` interface and `Map` interface defined in Table 1 and breaks them down to only ADD, GET and REMOVE events. It is also checked whether the collection exists in `ObjectIdentifier`. When the collection does not exist, a new allocation for the collection is added first and then an ADD for all already existing elements in the collection is added. This is necessary, because it is possible to create a object in Java without a constructor (i.e., for example through reflection) that is not catchable through AOP. The `ConstructorAspect`, `CollectionAspect` and `ObjectIdentifier` use the record methods in `Recorder` ④ to create a `DataRecord` that is written in `SaveToDisk` through a separate thread to the trace file. For that purpose, `Time` ⑤ saves the first time any pointcut is caught, which we use as start time of our application. Then, `Time` is used to calculate the time for the `DataRecord` of all other caught pointcuts and deallocations based on the saved start time, i.e., we report for every event its elapsed time since application start. Also, some information that `ConstructorAspect` and `CollectionAspect` save are texts which are often the exact same (for example method names or type names). It would take a lot of space to save these strings multiple times in the trace file. Thus, each string is assigned a unique number in `Symbols` ⑥ and is saved into a separate file called symbols. For each event, we then do not store the string but its symbol number, which can then be resolved using the symbol file during parsing.

4.2 Structure of the Trace File and Symbols File

The trace file is built on five different kinds of entries that are visualized in Figure 7. The first kind ① is a collection allocation with the specified *collectionID*, i.e., its `System.identityHashCode(Object x)`. The *size* represents how many objects are currently stored in the collection. This should typically be always zero, with the following exception: if an allocation was missed, its allocation event has to be recorded when another kind of event is encountered for said collection. As such, a different size for the collection can exist. The

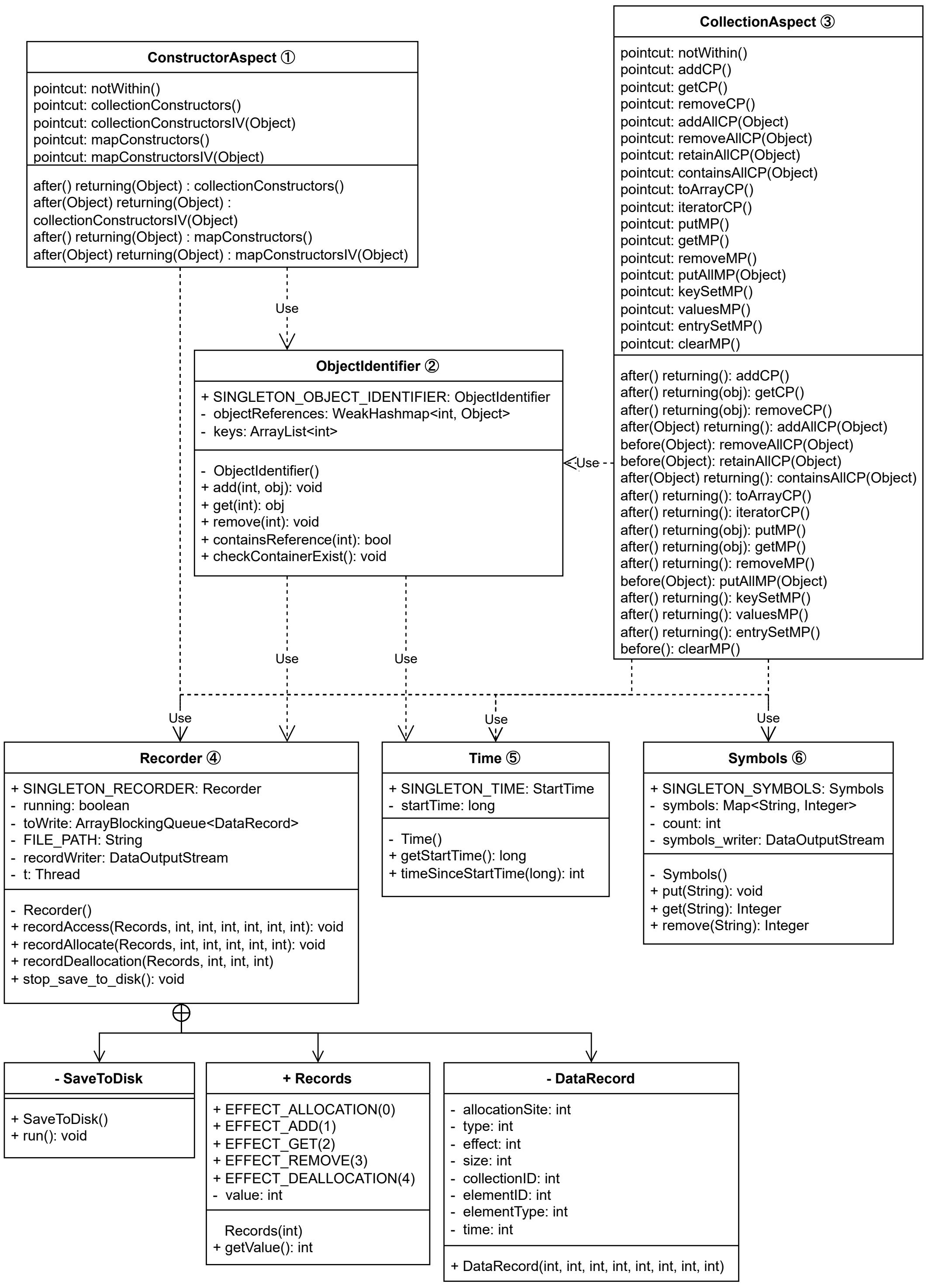

ConstructorAspect ①
pointcut: notWithin()
pointcut: collectionConstructors()
pointcut: collectionConstructorsIV(Object)
pointcut: mapConstructors()
pointcut: mapConstructorsIV(Object)
after() returning(Object) : collectionConstructors()
after(Object) returning(Object) : collectionConstructorsIV(Object)
after() returning(Object) : mapConstructors()
after(Object) returning(Object) : mapConstructorsIV(Object)
CollectionAspect ③
pointcut: notWithin()
pointcut: addCP()
pointcut: getCP()
pointcut: removeCP()
pointcut: addAllCP(Object)
pointcut: removeAllCP(Object)
pointcut: retainAllCP(Object)
pointcut: containsAllCP(Object)
pointcut: toArrayCP()
pointcut: iteratorCP()
pointcut: putMP()
pointcut: getMP()
pointcut: removeMP()
pointcut: putAllMP(Object)
pointcut: keySetMP()
pointcut: valuesMP()
pointcut: entrySetMP()
pointcut: clearMP()
after() returning(): addCP()
after() returning(obj): getCP()
after() returning(obj): removeCP()
after(Object) returning(): addAllCP(Object)
before(Object): removeAllCP(Object)
before(Object): retainAllCP(Object)
after(Object) returning(): containsAllCP(Object)
after() returning(): toArrayCP()
after() returning(): iteratorCP()
after() returning(obj): putMP()
after() returning(obj): getMP()
after() returning(): removeMP()
before(Object): putAllMP(Object)
after() returning(): keySetMP()
after() returning(): valuesMP()
after() returning(): entrySetMP()
before(): clearMP()
Use
ObjectIdentifier ②
+ SINGLETON_OBJECT_IDENTIFIER: ObjectIdentifier
- objectReferences: WeakHashmap<int, Object>
- keys: ArrayList<int>
- ObjectIdentifier()
+ add(int, obj): void
+ get(int): obj
+ remove(int): void
+ containsReference(int): bool
+ checkContainerExist(): void
Use
Use
Use
Use
Use
Use
Use
Recorder ④
+ SINGLETON_RECORDER: Recorder
- running: boolean
- toWrite: ArrayBlockingQueue<DataRecord>
- FILE_PATH: String
- recordWriter: DataOutputStream
- t: Thread
- Recorder()
+ recordAccess(Records, int, int, int, int, int, int): void
+ recordAllocate(Records, int, int, int, int, int): void
+ recordDeallocation(Records, int, int, int)
+ stop_save_to_disk(): void
Time ⑤
+ SINGLETON_TIME: StartTime
- startTime: long
- Time()
+ getStartTime(): long
+ timeSinceStartTime(long): int
Symbols ⑥
+ SINGLETON_SYMBOLS: Symbols
- symbols: Map<String, Integer>
- count: int
- symbols_writer: DataOutputStream
- Symbols()
+ put(String): void
+ get(String): Integer
+ remove(String): Integer
- SaveToDisk
+ SaveToDisk()
+ run(): void
+ Records
+ EFFECT_ALLOCATION(0)
+ EFFECT_ADD(1)
+ EFFECT_GET(2)
+ EFFECT_REMOVE(3)
+ EFFECT_DEALLOCATION(4)
- value: int
Records(int)
+ getValue(): int
- DataRecord
- allocationSite: int
- type: int
- effect: int
- size: int
- collectionID: int
- elementID: int
- elementType: int
- time: int
+ DataRecord(int, int, int, int, int, int, int, int)

Figure 6: Structure of the Aspects and Java classes that collect the data.

	Container Method Call	*Interpretation*
(a)	A.add(o)	ADD(A, o)
	o=A.get(..)	o=GET(A)
	o=A.remove(..)	REMOVE(A, o)
	A.addAll(B)	$\forall o \in B$, o=GET(B) $\forall o \in B$, ADD(A,o)
	A.removeAll(B)	$\forall o \in B$, o=GET(B) $\forall o \in A \cap B$, REMOVE(A, o)
	A.retainAll(B)	$\forall o \in B$, o=GET(B) $\forall o \in A \setminus B$, REMOVE($A$, o)
	A.containsAll(B)	$\forall o \in B$, o=GET(B)
	A.toArray()	$\forall o \in A$, o=GET(A)
	A.iterator()	$\forall o \in A$, o=GET(A)
	A.clear()	$\forall o \in A$, o=REMOVE(A)
(b)	v = A.get(k)	k=GET(A) if $v \neq$ null
	r=A.put(k, v)	ADD(A,k) if $r \neq$ null k=GET(A) otherwise
	r = A.remove(k)	REMOVE(A, k) if $r \neq$ null
	A.putAll(B)	$\forall k \in B.keySet()$, k=GET(B) $\forall k \in B.keySet()$: if $k \in A.keySet()$, k=GET(A) otherwise, ADD(A, k)
	A.keySet()	$\forall k \in A.keySet()$, l=GET(A)
	A.values()	$\forall k \in A.keySet()$, k=GET(A)
	A.entrySet()	$\forall k \in A.keySet()$, k=GET(A)
	A.clear()	$\forall k \in A.keySet()$, REMOVE(A, k)

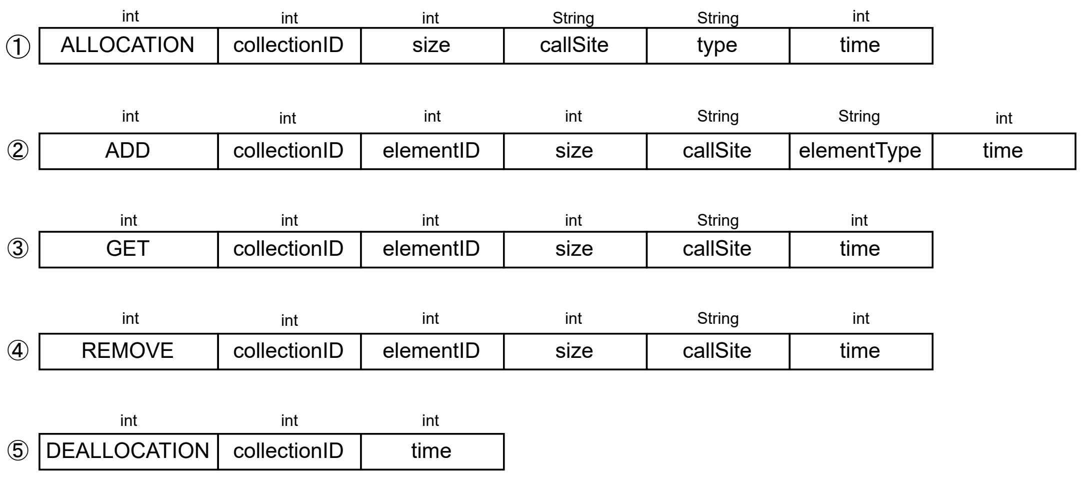
Table 1: (a) methods defined in java.util.Collection; (b) methods defined in java.util.Map. [30] For (a) the method clear was added in comparison to [30].

callSite describes the source location (file name + line number) where the allocation happened (or where the collection was first encountered, in the case we missed the allocation). As the collections can have different types (for example ArrayList or LinkedList), the name of the type is saved as *type*. Finally, the *time* of the allocation, relative to the *startTime* as described in Section 4.1, is saved. The second kind ② shows an addition to the collection. The *collectionID*, *size*, *callSite* and *time* are the same as in allocation. The new field *elementID* is the identifier, i.e., the `System.identityHashCode(Object x)`, of the element that was added to the collection. The *elementType* is the type of the element. The third kind ③ and the fourth kind ④ are similar to the second kind ②, but don't save the elementType. The third kind ③ shows the access of an element for the specified collection, while the fourth kind ④ shows the remove of an element for the specified collection. The last kind ⑤ shows the deallocation of the specified collection and at what time it was checked that it is deallocated.

As already mentioned above in Section 3 and Section 4.1, the strings are exchanged for unique int identifiers. This is done to reduce the file size, as the strings are frequently the exact same.

	Container Allocation Call	*Interpretation*
(a)	A=new Collection()	A=ALLOC Collection
	A=new Collection($initVals$)	A=ALLOC Collection, $\forall v \in initVals$, ADD(A, v)
(b)	A=new Map()	A=ALLOC Map
	A=new Map($initVals$)	A=ALLOC Map, $\forall v \in initVals$, ADD(A, v)

Table 2: (a) supported allocations for instantiable java.util.Collection implementors; (b) supported allocations for instantiable java.util.Map implementors.



① int ALLOCATION | int collectionID | int size | String callSite | String type | int time
② int ADD | int collectionID | int elementID | int size | String callSite | String elementType | int time
③ int GET | int collectionID | int elementID | int size | String callSite | int time
④ int REMOVE | int collectionID | int elementID | int size | String callSite | int time
⑤ int DEALLOCATION | int collectionID | int time

Figure 7: The initial design of the five different kinds of event entries in the trace file, where textual information such as *callSite* or *elementType* is stored uncompressed as a separate string in every event.

The size of a `String`, that is longer than two characters, is far larger than a single integer number in Java.

In Figure 8, the conversion through the *Symbols* file is visualized with the add kind. The first step is to identify which fields are strings. In the add kind, the fields *callSite* and *elementType* ① are both `String` data types. Both of them are added to the *Symbols* ② file when the exact string does not exist already. Then the *Unique Identifier* which is represented as an integer, is written to the trace file ③.

The trace file and symbols file are written as binary files to save file size. As the size of an int is 4 bytes [20] in Java, each kind can be easily read from the trace file. In contrast, the symbols file has a variable length per entry, as the Strings can vary in length. The solution for the variable length of Strings can be seen in Figure 8 *Symbols* ②. First, the *Unique Identifier* is saved. Second, the string length in bytes is saved, which is for the first entry 7 characters and the second entry 9 characters. As each character in Java has a size of 2 bytes [20], this means we actually save 14 bytes on the first entry and 18 bytes on the second one. Finally, the String

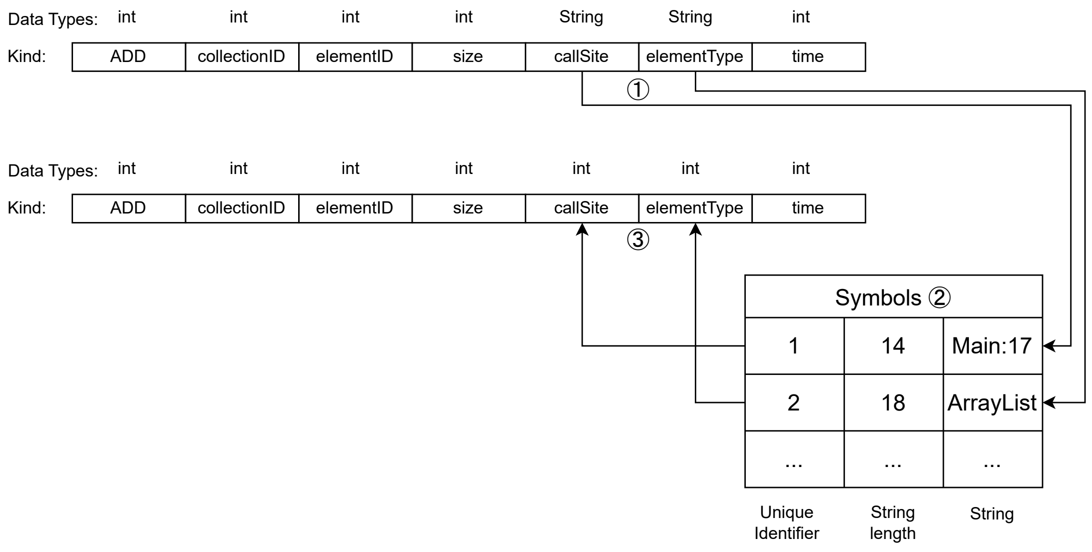


Figure 8: Changing String to unique int identifiers to reduce filesize.

itself is converted to bytes which can be read back in later through the use of the string's length.

Example Listing 3 shows a small Java program which creates ① an `ArrayList`, adds ② the element '1' to the `ArrayList` and accesses ③ that element after a short waiting period of 10ms. At the end, we add ④ another element. When we weave this program with the classes and aspects from Section 4.1, a trace file and a symbols file are created in binary form. These files are shown in human-readable form with '|' as a delimiter in Listing 4 and Listing 5. In the trace file, the first line ① is the allocation of the `ArrayList`, which is symbolized by kind 0. Then, the *collectionID* and the current size of the `ArrayList` follow. Following them is the *callSite*, which is substituted with 1 from the symbols file, which represents "Main:5". After the *callSite* is the type of the ALLOCATION. Looking up the type's number in the symbols file, we can find out that we allocated an `ArrayList`. At the end, the time of the ALLOCATION is written. Here, this ALLOCATION is the first entry, and as such, it sets the start time and gets the value 0. The second line ② is an ADD, represented by the kind 1. The *collectionID* is the same as in ALLOCATION, and we have the *elementID* from the added element. Because of the ADD, the size of the list is now 1. Then we have the *callSite* and *elementType*, which we have to look up in the symbols file for their `String` values. At the end, the time slightly elapses compared to the ALLOCATION. But it could also have the same time, if the execution was done in the same millisecond. The third line ③ is a GET and extremely similar to the ADD, except it has no *elementType*. The *collectionID*, *elementID* and *size* are the same, as we call the previously added element from a different *callSite*. The time here jumps from 1ms to 11ms, as we have in the code a waiting period of 10ms. The fourth line ④ is another ADD with a different element. After comparing this second ADD to the first, we find that the *elementType* is the same number. This is the case, because we have the same type of element, and we can reuse the already saved `String` in the symbols file.

```java
import java.util.ArrayList;

class Main {
  public static void main() {
    ArrayList<Integer> myList = new ArrayList<>(); ①
    myList.add(1); ②
    Thread.sleep(10);
    myList.get(0); ③
    myList.add(2); ④
  }
}
```

Listing 3: Simple program to create an ALLOCATION, ADD and GET on an `ArrayList`.

```
0|285377351|0|1|2|0               ①
1|285377351|156727562|1|3|4|1     ②
2|285377351|156727562|1|5|11      ③
1|285377351|146123415|2|6|4|12    ④
```

Listing 4: Generated trace file in human readable form with | as delimiter.

```
1|12|Main:5
2|18|ArrayList   ①
3|12|Main:6
4|14|Integer     ②
5|12|Main:8      ③
6|12|Main:9      ④
```

Listing 5: Generated symbols file in human readable form with | as delimiter.

5 Backend

After the generation of the trace file and the symbols file, the backend introduces the server that hosts the frontend and creates the parser for data processing.

Server Structure Figure 9 shows the responsibilities of the *server* ①. When the *server* ① is started, the *frontend* ③ will be hosted. Because of this, it is possible to interact with the server through a browser(i.e., a local running web application). Depending on the interaction from the user in the frontend, the frontend will request specific data from the *server* ①. For example, when the user requests the processing of a selected trace file and a selected symbols file in the *frontend* ③, the *server* ① will in response create a *parser* ② instance which processes the files. Afterwards, the requests will be piped through the *server* ① to the *parser* ② instance and the returned data (e.g., the staleness of collections) will be sent through the *server* ① to the frontend.

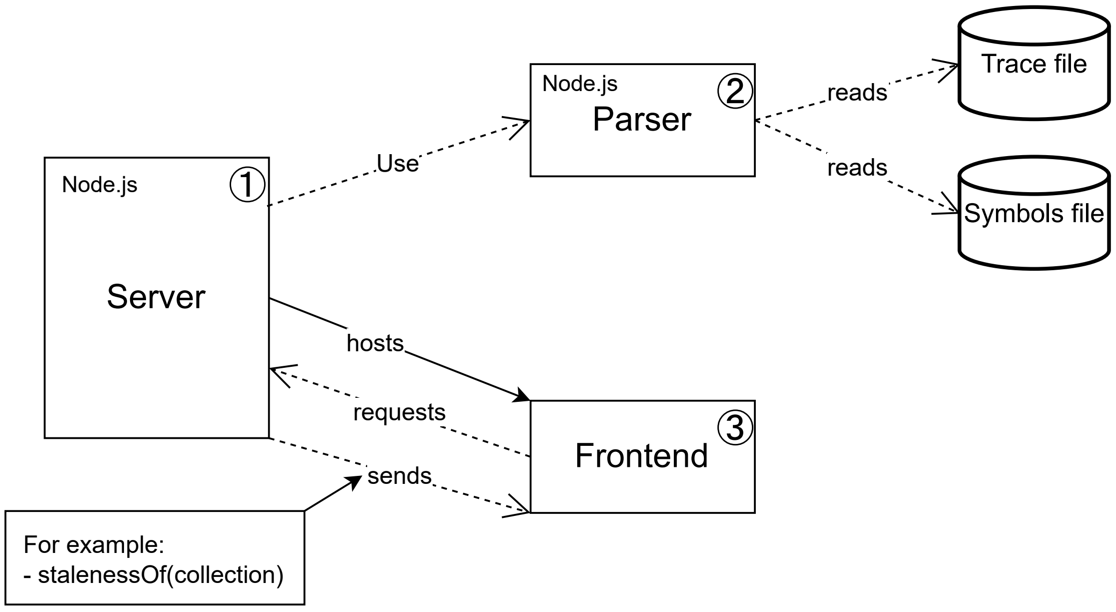


Figure 9: The backend for the communication between the frontend and the parser.

The server is needed to host the frontend but the extraction of the parser from the frontend was a design decision. The runtime environment *node.js*, which the parser and server use, allows for better file processing and easier debugging then the native javascript in the browser.

6 Parser

After the backend, the parser that handles the file reading and the calculations will be introduced in more detail.

6.1 Parser Structure

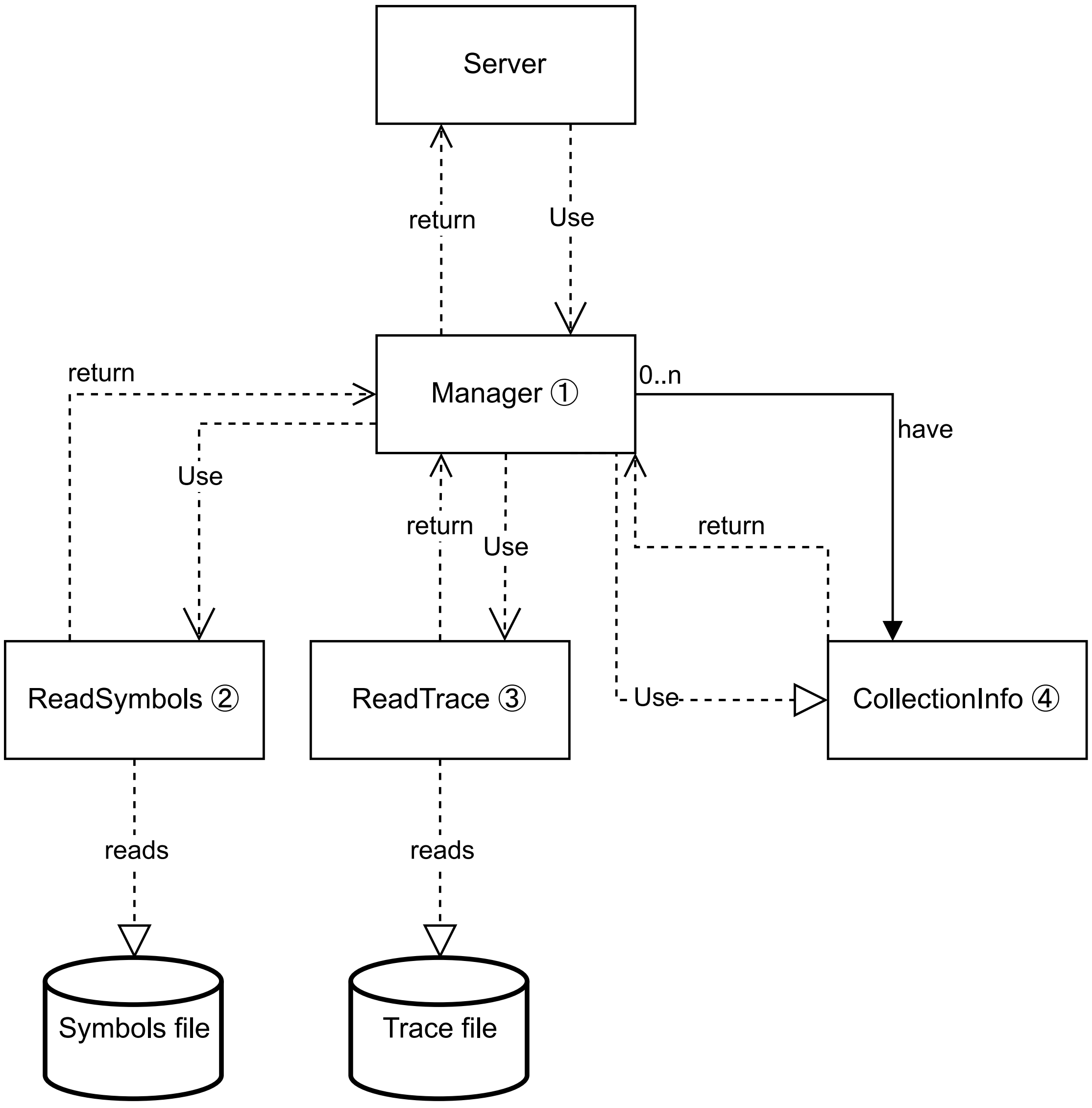


Figure 10: Simplified structure of the parser to process the trace file and symbols file.

In Figure 10 a simplified structure of the parser is illustrated. The following list introduces the purpose of each part of the parser.

① Once the `Manager` is created, `ReadSymbols` and `ReadTrace` are created with the filepaths received from the frontend. The data received from `ReadTrace` is handled through the kind of the data. When an ALLOCATION event is received, a `CollectionInfo` object will be created. For ADD events, GET events and REMOVE events, the corresponding `CollectionInfo` object is updated. When a DEALLOCATION event for a collection is received, the time for the DEALLOCATION event will be set in the respective `CollectionInfo`. When a request from `Main` occurs, the necessary data from the collections is gathered and returned as a JSON String to `Main`.
② The `ReadSymbols` class reads the symbols file and saves the data in a map. It then provides access to the map.
③ The `ReadTrace` uses the listener pattern to allow clients to process the read data. For that reason, the reading of the trace file needs to be explicitly started, after listeners, i.e. Manager, have been be added. After the start, it reads the file event by event, converts it into an event object and then informs all listeners.
④ In `CollectionInfo`, all accesses read from the trace file for one specific collection are saved. Also, the calculations for that specific collection happens here. The calculations include for example the staleness calculation and the aggregation, i.e., the summation, of the elements that are added, accessed and removed from the same location.

6.2 Staleness Calculation

The staleness of an element in a collection is defined as t_2 - t_1, where the element is removed at t_2 and the last access happened at t_1. When the element is never removed, t_2 is set to the deallocation of the collection. When the remove or deallocation happens after the ending time of the viewed region t_e, then t_2 is set to t_e. When there is no access for setting t_1, then t_1 is set to the time when the element was added to the collection. Should the last access happen before the start time of the viewed region t_s, then t_1 is set to t_s. To make the staleness values of elements better comparable, the staleness is normalized to a value between 0 and 1, where 0 is the least stale and 1 is the most stale. This is done by dividing (t_2 - t_1) by (t_e - t_s). For the whole collection, the staleness of all elements are added and then divided by the amount of elements. [30]

Figure 11 shows the different possibilities that occur when calculating the staleness for one element. But it must be mentioned, that for simplicity, only removes and no deallocations were used in the graphic, but wherever a remove is done, a deallocation of the collection could also have happened.
① The basic choice for t_1 and t_2, where t_1 is the last access through a GET and t_2 is the REMOVE of the element from the collection.
② Here, t_1 is set to be the ADD time of the element, as no access happened between the ADD and the REMOVE. t_2 is here also set to the REMOVE.
③ The REMOVE happened after the end time t_e, which means we have to set t_2 to t_e. t_1 will be set to the last access, which happens to be here a GET.
④ The last access, which is here the adding of the element, was before our start time t_s. As such, we need to set t_1 to t_s and t_2 to the remove that happened in the viewed region.
⑤ This is the worst case possible, as the last access happened before the start time t_s and the remove happened after the end time t_e. Because of that, we need to set t_1 to t_s and t_2 to t_e.

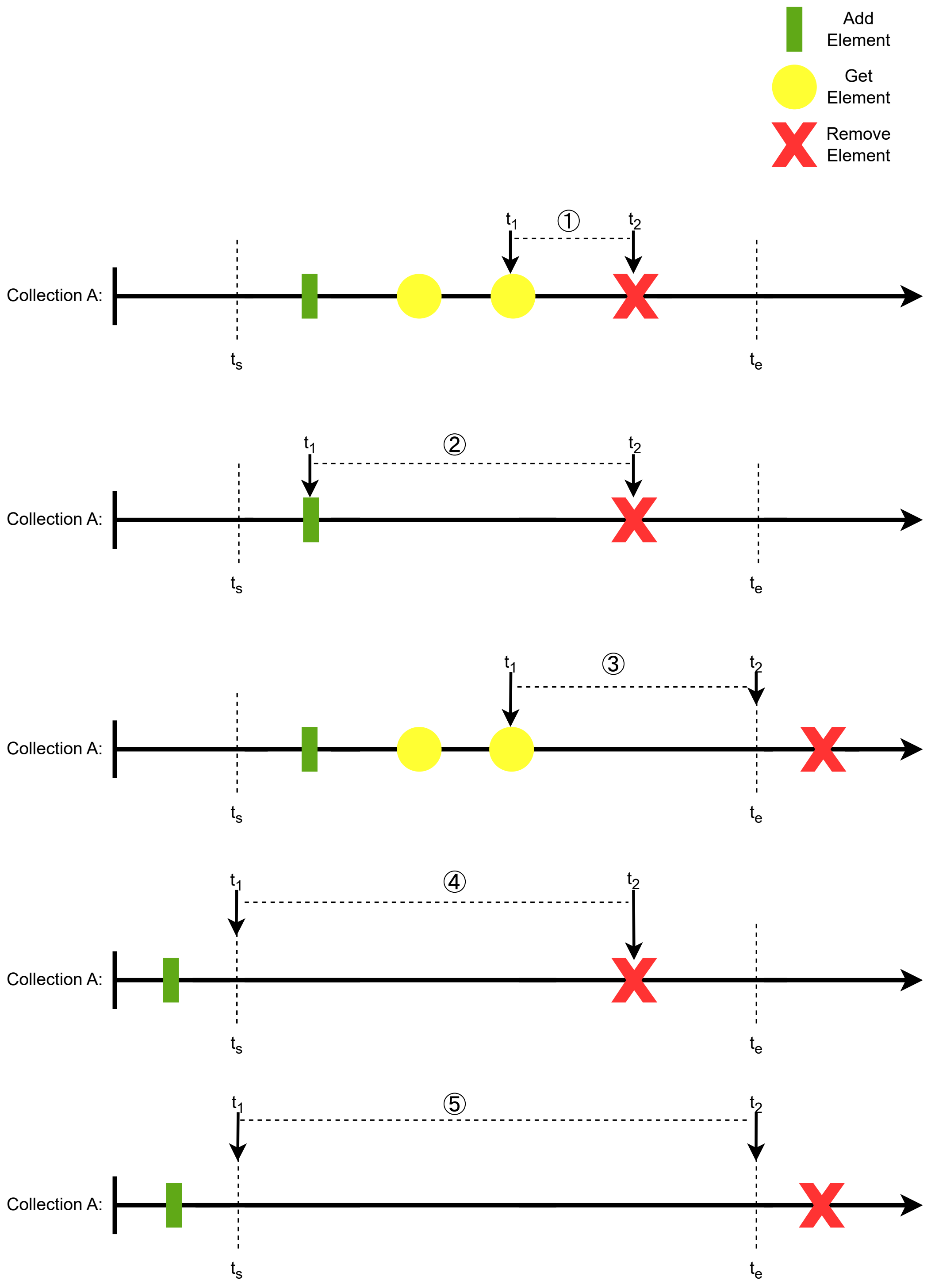


Figure 11: The different possibilities to calculate the staleness of one element in a collection.

This means that the element exists through the whole viewed region. Therefore, the elements staleness value is the maximum possible amount for this region, i.e., 1.

Example Figure 12 illustrates the calculation of the staleness of a collection with only two elements.

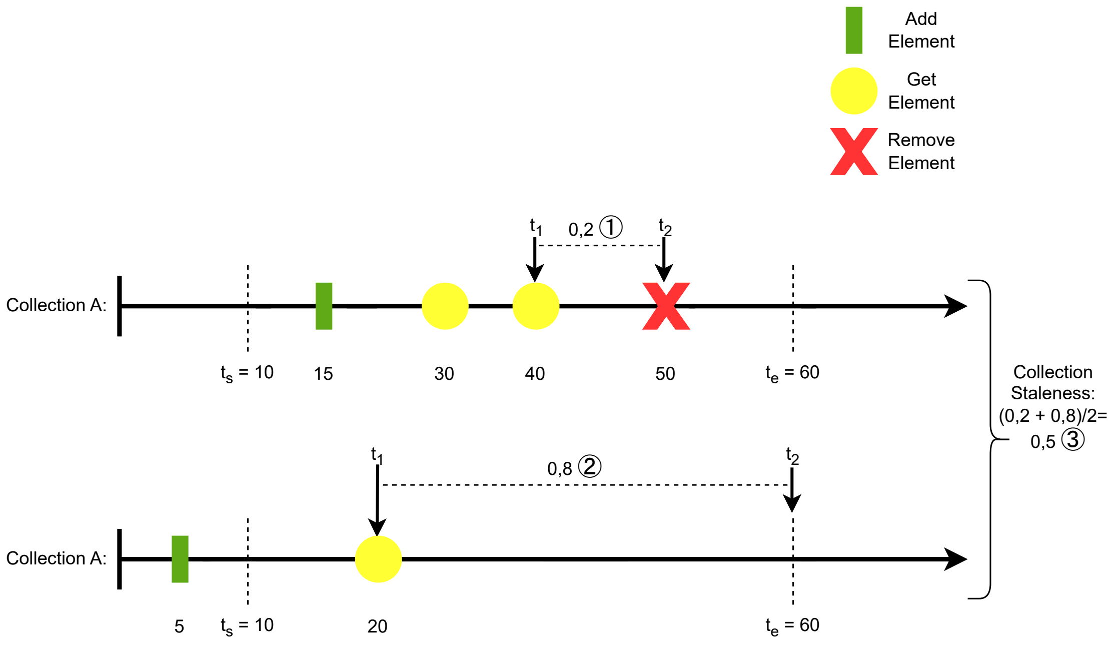


Figure 12: Staleness of a collection with two elements.

In Figure 12, the first element was added at time 15, was accessed at time 30, was accessed at time 40 and finally removed at time 50. We view the region from the start time $t_s = 10$ to the end time $t_e = 60$. As the element was removed at time 50, the element was removed in the viewed region. Therefore, we set $t_2 = 50$. The last access was the GET at time 40, which means we set $t_1 = 40$. Now we calculate the staleness for this one element, which means we fill in the values for the formula $\frac{t_2-t_1}{t_e-t_s}$. That results in $\frac{50-40}{60-10} = \frac{10}{50} = \frac{1}{5} = 0,2$ ①. The second element has the same t_s and t_e, but the last access was at time 20 and the remove did not happen at all. As such, we get $t_1 = 20$ and $t_2 = t_e = 60$. Now we insert this also in the formula and get $\frac{60-20}{60-10} = \frac{40}{50} = \frac{4}{5} = 0{,}8$ ②. We have now calculated the staleness value for the only two elements in collection A in the viewed region. Therefore, we can now calculate the staleness for the whole collection A by adding the calculated staleness of the elements in the viewed region. Then, divide the sum by the amount of elements to get the average staleness of the whole collection in the viewed region. When inserting the values, we get the average staleness $\frac{\sum \text{element staleness}}{\text{amount of elements}} = \frac{0{,}2+0{,}8}{2} = \frac{1}{2} = 0{,}5$ ③ as a result for the staleness of collection A.

6.2.1 Problem of Duplicate Elements

The approach of calculating the staleness for each element has problems. One such problem is that we can have the same identifier for two elements in a collection. After all, it is allowed to add the same element multiple times in some collection types such as lists.

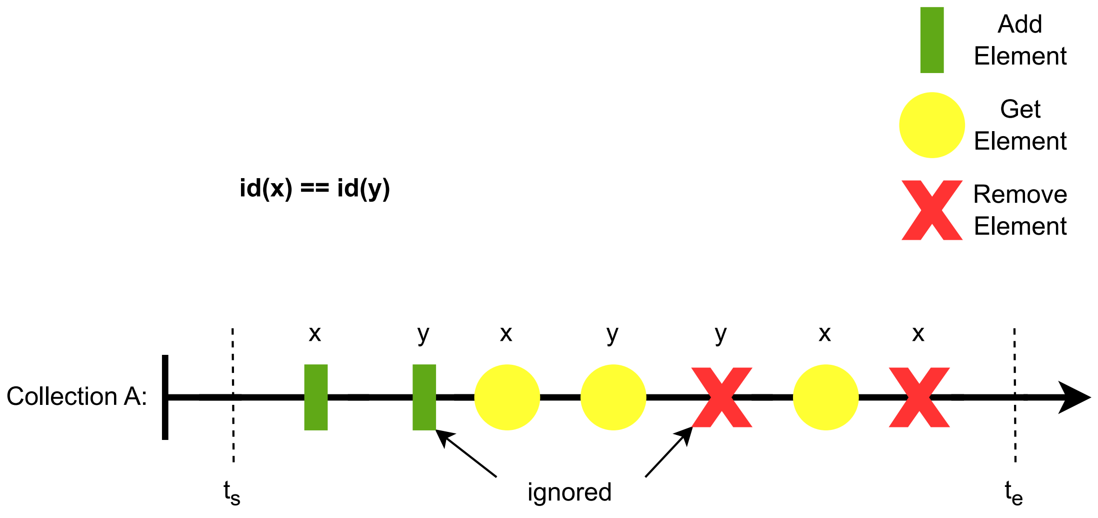


Figure 13: Problem: Having multiple elements with the same identifier (x and y) in the same collection at the same time.

Figure 13 visualizes this problem, where element x and y have the exact same identifier. Therefore, it is impossible to match the operations to the correct element. To solve this problem, the ADDs and REMOVEs, except the first ADD and the last REMOVE, are deleted. This leaves in Figure 13 the ADD and REMOVE of x. Then, all accesses between the remaining ADD and the remaining REMOVE are handled, as if the accesses are for one element.

This solution distorts the results of the staleness calculation. But normally, there should not exist too many duplicates at the same time in larger collections, so that the overall staleness would not be distorted by much.

6.2.2 Problem of Reinserting Elements

Another problem that may occur for the parser is reinserting an already removed element into the same collection. The problem by reinserting elements is the associated data which is identified by the element identifier. The reinsert problem is visualized in Figure 14. We see that there is no crosscutting between the two operations. Therefore, the solution is simply renaming the reinserted element. In this work, it was chosen to add a '+' to the end of the identifier.

6.3 Aggregation of the Elements

We want to be able to identify the locations of a collection where ADD events, GET events and REMOVE events occurred. For this reason, it is necessary to aggregate GET events and

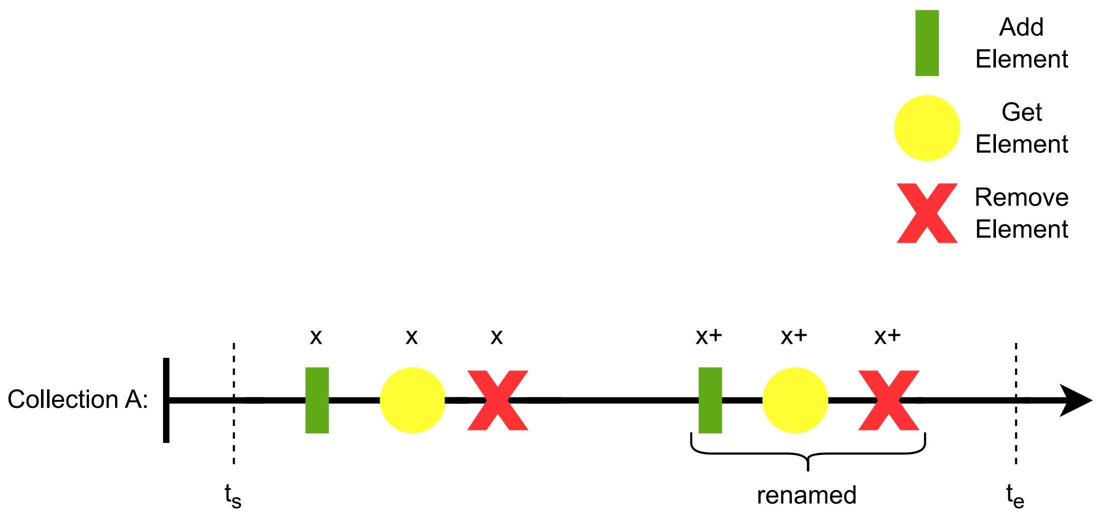
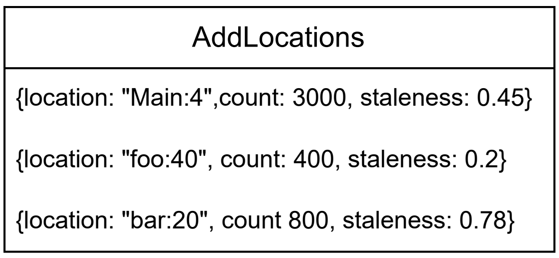


Figure 14: Problem: Reinserting the same element after removing the element from the collection.

REMOVE events and save for each element the ADD location, i.e., the location where the element was added to the collection. We achieve this by saving each encountered GET location (i.e., the location where the elements were accessed) of a collection and each encountered REMOVE locations (i.e., the location where the elements were removed from the collection) of a collection into a map and aggregate the values when a GET event or a REMOVE event for that specific collection occurred. The ADD location is saved directly with the element, so that it is possible to associate the calculated staleness value of the element with the ADD location.



AddLocations
{location: "Main:4",count: 3000, staleness: 0.45}
{location: "foo:40", count: 400, staleness: 0.2}
{location: "bar:20", count 800, staleness: 0.78}

Figure 15: The array of JSON Strings that is created for ADD locations.

When requested, each location data is transformed into a JSON String and then added to an array that is returned. Figure 15 illustrates how this transformed state looks for the ADD locations. First, the location where the elements were added is provided. Second, the amount of elements added at that location is given. Finally, the average staleness of the added elements in the selected time frame will be send. In Figure 15 we have three different locations, "Main:4" with 3000 elements and an average staleness of 0.45, "foo:40" with 400 elements and an average staleness of 0.2 and "bar:20" with 800 elements and an average staleness of 0.78. For GET locations and REMOVE locations it works similar except that there will be no average staleness of the elements calculated. For example, it seems not useful for GET locations, as the same element could be accessed multiple times from the same location.

7 Frontend

Now that the the symbols file and the trace file were processed, we explore in this chapter how the processed data is visualized.

Example The visualization is explained best through an example, as such the trace file and symbols file created from the example code in Listing 6 will be used throughout this chapter for the creation of the charts.

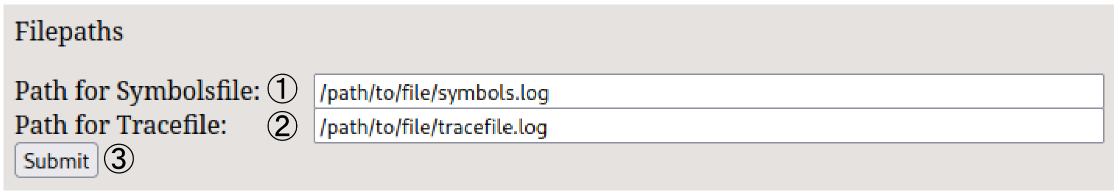
```java
import java.util.*;
public class Main {
    public static void main(String[] args) throws Exception {
        ArrayList<Integer> list = new ArrayList<>();
        Map<Integer, Integer> map = new HashMap<>(); ①
        for(int i = 0; i < 1000; i++) { ②
            list.add(i);
            map.put(i, i);
        }
        Thread.sleep(100); ③
        for(int i = 0; i < 500; i++) { ④
            System.out.println(list.get(i));
            System.out.println(map.get(i));
        }
        Thread.sleep(100);
        for(int i = 1000; i < 1100; i++) { ⑤
            list.add(i);
        }
        Thread.sleep(100);
        for(int i = 0; i < 100; i++) { ⑥
            System.out.println(list.get(i));
        }
        Thread.sleep(100);
        int n = 0;
        for(int i = 800; i < 900; i++) { ⑦
            System.out.println(list.remove(i - n));
            System.out.println(map.remove(i));
            n++;
        }
        Thread.sleep(100);
        System.out.println("List: " + list.size());
        System.out.println("Map: " + map.size());
    }
}
```

Listing 6: Example Java program that will be used for the visualizations of the frontend.

Listing 6 first allocates ① an `ArrayList` and a `HashMap`. Then, 1.000 elements are added ② to both collections. For better visibility in the charts, a waiting time ③ of 100ms is added between each operation. Then, the first 500 elements are accessed ④ once for each collection. To showcase a second add location, another 100 elements are added ⑤ to the `ArrayList`. In addition, to also have elements with a different amount of accesses, the first 100 elements of the `ArrayList` are accessed ⑥ a second time. At the end, both collections remove ⑦ 100 elements from the collection.

7.1 User Interface

At the beginning, the user interface asks the user for filepaths to the symbols file and to the trace file. Figure 16 illustrates how this looks, when visualized. The path to the symbols file ① and the path to the trace file ② have to be entered starting from the root directory. It is done this way, because for privacy reasons a file chooser would give a fake path to the file name. Also, *node.js* has better file processing capabilities then native Javascript in the browser. Furthermore, the complexer methods would be too time-consuming for this project. Finally, the submit button ③ sends the file paths to the parser.



Filepaths

Path for Symbolsfile: ① /path/to/file/symbols.log
Path for Tracefile: ② /path/to/file/tracefile.log
Submit ③

Figure 16: Choosing the paths of the symbols file and the trace file for the parser.

7.1.1 Element Size

After the submission the parser will start reading the files. The parser sends a progress information for every ten percent of the trace file read. When the trace file is completely read, the parser will send one hundred data points equally distributed over the entire captured program execution time. One data point contains the element size of all collections at the data points specified time.

Figure 17 shows the loading progress of the trace file, the line chart generated from the data points and the choice of the time frame for which the calculations shall be done. The following list describes the interesting parts of Figure 17 in more detail.
① The loading progress bar is filled in ten steps. Each step is executed when the parser sends that ten percent more of the trace file was read.
② In this part of the line chart, we see the 2.000 elements that were added to the two collections in Listing 6 ②.
③ This small increase symbolizes the 100 elements added in Listing 6 ⑤. When the increase of elements or decrease of elements is too small to be seen, then hovering over the points on the line will show the element size and the time of that data point.

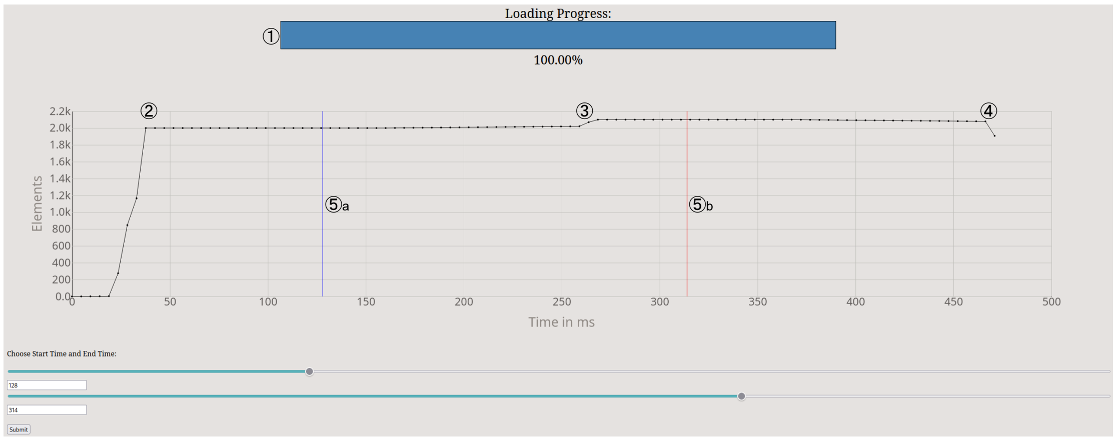


Figure 17: Overview of the data read from the trace file and symbols file generated from Listing 6.

④ The 200 elements removed at the end in Listing 6 ⑦ is visualized here. When looking closer, a slight decrease of elements happened at first, before the bulk of the elements was removed at once.
⑤ The blue line (⑤a) is the start time for the calculation time frame and the red line (⑤b) is the end time for the calculation time frame. The times can be chosen with the sliders below where the first slider is for the start time and the second slider is for the end time.

For the rest of the charts below in Section 7.1, the start time was chosen as 0 and the end time was chosen as the maximal available value of 471ms. These values were than sent to the parser through the submit button.

7.1.2 Collection Staleness

After selecting the calculation time frame and sending it to the parser, the parser calculates the staleness of the collections. The staleness data is then displayed in a bar chart where at most ten collections at the same time are displayed. The limit of a maximum of ten collections was chosen for all bar charts, because there could be more than 1.000 bars, and it would become hard to read.

The ordering of the staleness chart is based on the size of the collection multiplied with the average staleness value of the collection. Only using the average staleness value for ordering has the risk that only small, very stale collections are shown. Larger collections that could leak would then be basically ignored.

Figure 18 shows this bar chart with two sliders that allow the user to change the staleness threshold as well as the maximal size of the displayed collections.
① The staleness threshold changes in the bar chart the distribution of the blue and red parts. The blue part of the bar represents the numbered elements of the collections which have a staleness value below the threshold or equal to the threshold. The red part of the bar represents the numbered elements of the collections which are above the threshold.
② The maximum size changes the displayed collections. When there are collections that could not be displayed because of the limit, the slider allows to restrict to a lower maximum size. This

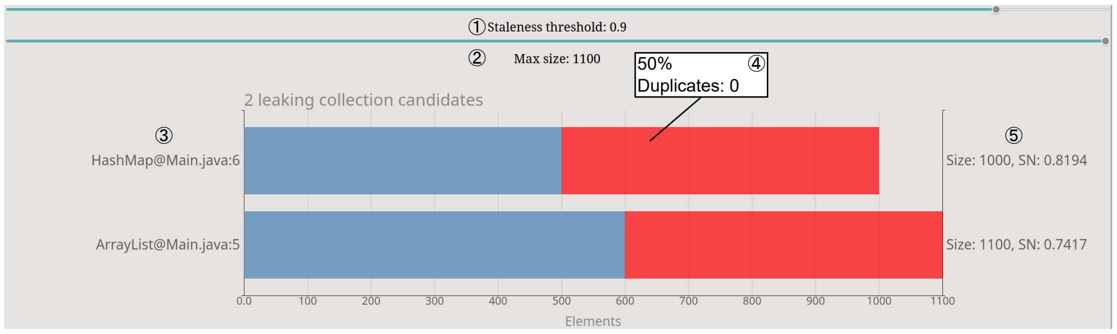


Figure 18: Staleness bar chart for collections over the whole program execution of Listing 6.

allows to also analyze smaller collections after we analyzed the larger ones. In our example the `ArrayList` with 1.100 elements is the largest, which means 1.100 is our maximum size that we can choose. When we change the slider to display only collections with 1.050 elements or below 1.050 elements in Figure 18, we will only be able to see the `HashMap`.

③ The type of the collection and in which file on which line it was allocated is displayed on the left. There can be multiple collections with the same type and allocation location. But even then, when there is a possible leaking collection it can be reduced to that specific location.

④ The text displayed when hovering over the bar. The staleness in percent of the colored part of the bar is displayed as well as the amount of duplicates in the whole collection.

⑤ The right text displays the element amount of the collection as size, as well as the average staleness value (SN) for the whole collection.

We acquire from this bar chart a first view of the different collections that are memory leaking candidates and how frequently their data is accessed. Based on this, the less frequently accessed collections are more likely to leak and should be viewed more closely by clicking on the bar of the collection.

7.1.3 Operations Count

After clicking on a bar, more charts specifically for the selected collection are shown. The first chart displays the number of ADDs, GETs and REMOVEs, as shown in Figure 19. The type and ALLOCATION location ① of the chosen collection is displayed on the top. The chart below displays the amounts of ADDs ②, GETs ③ and REMOVEs ④. At first glance we see that we picked to view the `ArrayList` from Listing 6. The `ArrayList` had a lot more ADDs than GETs. Also, only 100 elements were removed from the collection at all. This shows that we had added elements in Listing 6 that were never accessed and only wasted memory space until the end of the program.

In this chart we would have ideally way more GETs than ADDs. This would then mean that the data in the collection is far more often accessed and it is more unlikely that we have elements that only waste memory space.

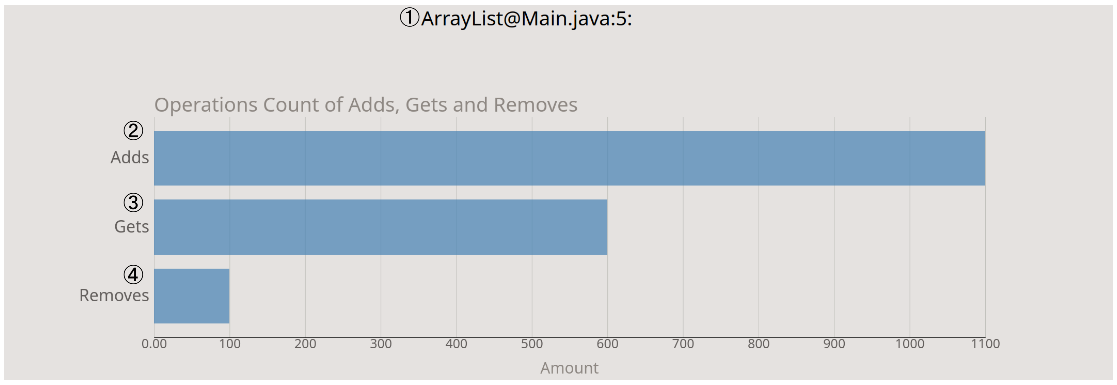


Figure 19: Displays the amount of ADD, GET and REMOVE operations for the `ArrayList` from Listing 6.

7.1.4 Element Get Count

Right afterwards we take a closer look at how the accesses were distributed between the elements. When there are a lot of elements with zero accesses, even when we have a lot more accesses than ADDs, the collection becomes a prime candidate for leaking memory.

Figure 20 shows how frequently elements were accessed. The *max get count* slider ① allows changing which counts are displayed in the bar chart. The elements above the chosen value that are not displayed will add up every access and then shown it as *count others* ④. Here, that are the 100 elements in Listing 6 ⑥ that were accessed a second time. In addition we have 600 elements that were never accessed ② and 400 elements that were accessed once ③. If we add the 400 elements with one access and the 100 elements with two accesses, we get the 600 accesses from the operation chart in Figure 19.

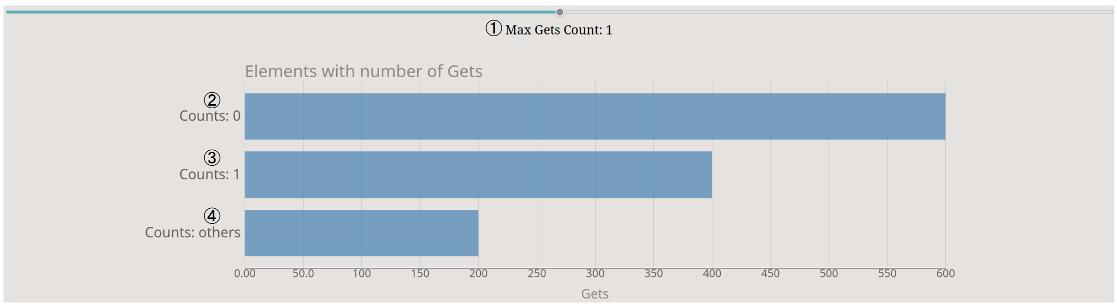


Figure 20: Elements with different amounts of accesses.

Through this chart it is possible to better see the accesses of the elements and find problems such as a lot of not accessed elements or only a few highly accessed elements while most elements were only accessed a few times.

Now that we viewed the operation counts and the access counts it would be interesting to know from which locations in the code the adding, accessing and removing was done.

7.1.5 ADDs per ADD Location

This section explains our "ADD per location" feature. It counts all ADD events that happened for each specific line of code in the files of the traced program for our viewed collection. The result will be displayed in a bar chart where the maximum amount can be changed through a slider. This is shown in Figure 21 where we have the slider ① for the maximum size for ADD locations at the top. The first ADD location ② in the bar chart and the second ADD location ③ in the bar chart are the locations in Listing 6 where elements were added to the `ArrayList`. It is also possible to display the average calculated staleness of the elements ① for each line as the color of the bar and display it in percent when hovering over the bar. The redder the bar is, the more stale the elements added at this location are. In contrast, the bluer the bar, the less stale the elements added at this location. Here, we see that the first line ② has more stale elements added than the second line ③.

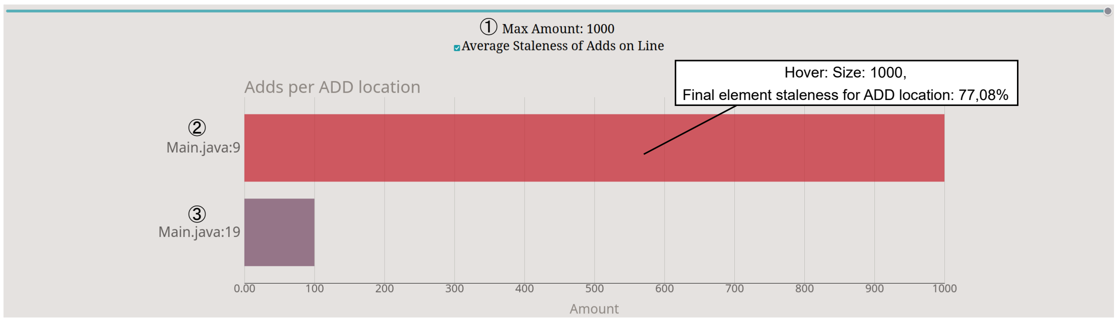


Figure 21: The location in the code where the elements were added to the collection. The bar color shows the average element staleness for the ADD location.

The ADD locations show a nice overview from where in the code the elements of our viewed collection were added. And through the feature of showing the average element staleness of our time frame as color and in percent when hovering over the bar, it is easily distinguishable from where the stale elements were added to the collection.

7.1.6 GETs per GET Location

Second, we count all accesses for each specific line of code in the program. The resulting chart shown in Figure 22 has the same structure as the chart in Figure 21. Only the function to show the average element staleness is missing because it would make little sense. After all, it is possible that the same element was accessed multiple times from this location or in an extreme case, the same element was only accessed from this location. As such, the average staleness for the elements would have little value.

This chart allows for a better understanding where in the code the accesses to the viewed collection occurred. The most accessed locations may be interesting for optimization while the least accessed locations, when not done on purpose, hold a higher potential for an error.

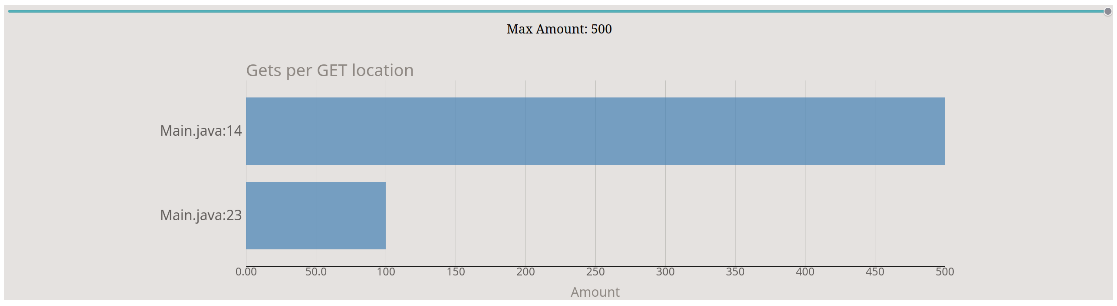


Figure 22: The location where the accesses of the viewed collection occurred.

7.1.7 REMOVEs per REMOVE Location

Finally, we have the chart were the removes of the viewed collection were counted. The chart in Figure 23 has the same structure as the chart in Figure 22. What is different is that only one line in Listing 6 removed elements from the viewed collection. As such, the bar takes up more space as it does not need to share the space with other lines.

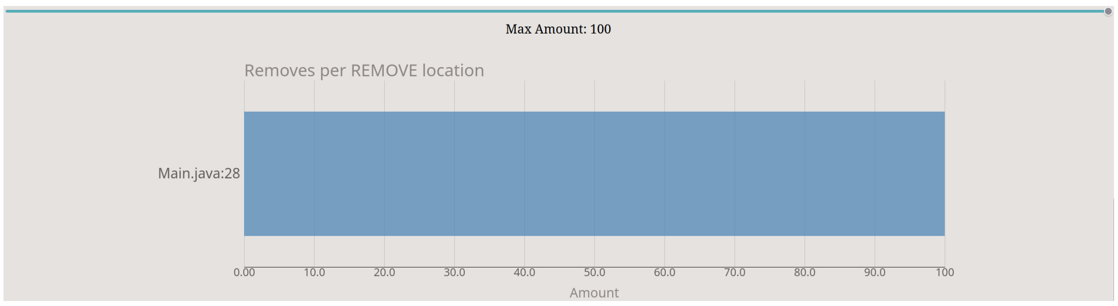


Figure 23: Remove locations of the viewed collection.

The REMOVE locations of the elements of the viewed collection allow checking if the program works as anticipated. It gives a lead where to start the search in the program, when there are a lot of removes at a not anticipated location.

All charts together provide a lot of leads for possible memory leaks. But looking closer, they also allow to detect patterns of inefficient memory consumption.

7.2 Anti-Patterns of Memory Inefficiency

The larger the tested applications become, the more often they suffer from excessive memory consumption. This mostly happens because common design mistakes are made. For example, a collection is allocated through the standard constructor in Java and no elements were added. Then, there was a specific amount of memory space allocated for the overhead of managing the collection and an initial capacity for the references to the elements. This results in a fixed cost in memory space while no elements were added to the collection. Such common mistakes are mostly easily understood and easily fixable. Therefore, the common mistakes can be summarized in patterns that should be avoided or in short, anti-patterns. The example of the empty collection above is as such called the empty collection pattern. [5]

Chis et al. [5] introduced a few different patterns, that help to reduce unnecessary consumed memory. A few of the more interesting patterns for this work will be introduced in more detail in the following list.

1. *Empty collection:* A completely empty collection that was allocated and had no elements added over the program execution. Depending on the type of the collection, there is a different fixed cost that is consumed regardless. For example, according to Chis et al. [5] is in the Java standard library on a 32-bit JVM the fixed cost of a `HashSet` around 100 bytes of memory while a `ConcurrentHashMap` has a fixed cost of around 1600 bytes.

2. *Small collection:* Small collections have a few elements added. But the overhead for managing the collection takes up a large portion of the memory used by the collection. This gets even worse if there is unused allocated memory space for elements.

3. *Fixed collection:* The fixed collection describes a special case of a small collection where always the exact same number of elements exists in the list. When there are only a few elements, the overhead and possibly the initial capacity take up more memory space than the elements themselves. A better solution for such a usage would be an array instead of a collection.

4. *Sparsely populated collection:* Most array-based implementations of collections have a tendency of being sparsely populated. This means, that only a few entries from the overall allocated entries point to elements. This is shown in Figure 24 where the first and the seventh entries contain a reference to an element. The other eight entries are empty.

   There are different causes why this can happen: 1) the initial capacity of the collection is too large, or 2) after removing a lot of elements, the collection was not trimmed-to-fit, or 3) the collection's growth policy is way too aggressive and starts allocating new memory space while there are still a lot of entries empty.

5. *Boxed scalar collections:* In Java the standard collections do not support primitives for keys, values or entries. This results in boxing up the primitives into wrapper objects that consume more memory than the actual data. This wrapping also happens in Listing 6 ① where `Integer` had to be used in the collections instead of the primitive `int`.

6. *Nested collections:* As the name nested collections implies, this pattern describes the common case of collections added as a whole to collections. For example, such a case would be `HashMap` with `HashSets`, so that multiple values per entry key can exist. Generally, the problem here is when the outer collection is significantly larger than the inner collections and a larger overhead cost exists. Switching the outer collection with the inner collection can reduce collections used and as such it is possible to save a significant amount of memory.

From this list, the first four anti-patterns describe problems that occur to the initial allocated capacity or where the overhead takes up a significant part of the memory footprint. This can mean that the standard constructor allocated more memory than actually needed or that the user specified a larger initial capacity than actually needed. The other two anti-patterns describe

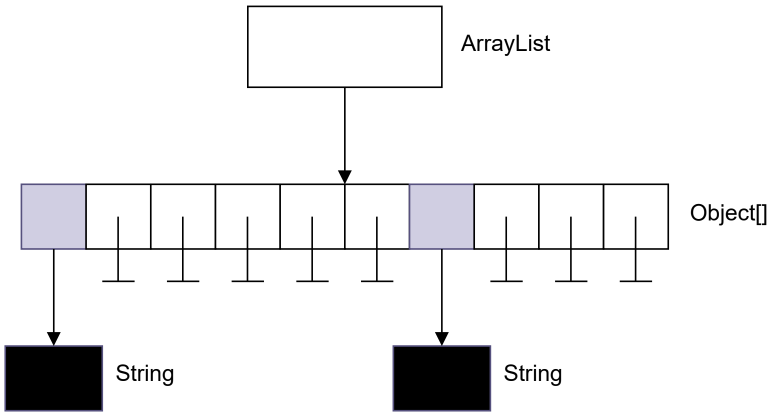


Figure 24: An example of the *sparse collection* pattern, in this case with eight null slots inspired from Fig. 3 in [5].

more specific problems. From the described anti-patterns by Chis et al. [5], this six show the most potential to be applied to the approach of this work.

7.2.1 Implemented Anti-Patterns

From the listed anti-patterns above, the first anti-pattern, i.e., *empty collection* pattern and the second anti-pattern, i.e., *small collection* pattern are fully functional. The collections can be filtered in Figure 18 for collections at a specified size or below a specified size. As such, when the maximum size is set to zero, empty collections will be listed and show where they were allocated in the code. In the same way it is possible to filter for small collections.

However, the *fixed collection* pattern is harder to identify. The user would need to check the same collection over different time frames. The checked collection should have the same collection size at all of them. But even then it can not be guaranteed that the collection is really fixed to that size or if the checked time frames were unluckily chosen. Unfortunately, the time schedule for this thesis did not leave more time to implement anymore anti-patterns than this three in a meaningful manner. Nevertheless, the tool resulting from this work allows more visual inspections based on staleness then originally anticipated.

8 Evaluation / Usage

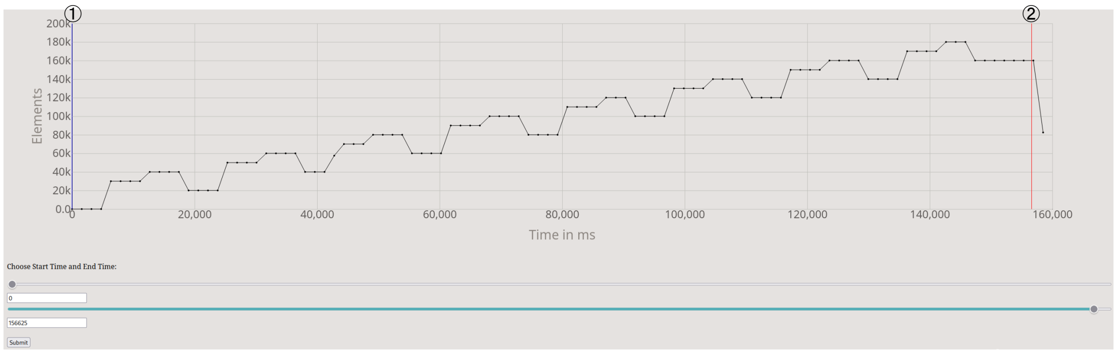
After introducing the approach in more detail, let us evaluate how well it works by comparing it to the results of another work.

Commons HttpClient Weninger et al. [27] did an evaluation on the leaking version 3.0.13 of the Commons HttpClient library written in Java with their created tool. The Commons HttpClient library is a library that can be used to send HTTP requests. As such, a small driver application that creates HTTP connections in multiple batches was developed by Weninger et al. [27]. Each batch creates 10.000 connections and deletes them shortly afterwards. With their developed tool they saw a continuous memory growth instead of only spikes of memory usage. After analyzing the problem more deeply with their tool, they found that objects in `HostConnectionPool` were kept alive by a `HashMap`. The tool provided also the name of the method in which the `HostConnectionPool` objects were allocated which gave them enough information to investigate the problem on the source code level. The result of that investigation was that the `HostConnectionPool` objects were added to the `HashMap` but never deleted from it, which resulted in the memory leak. [27]

This problem of leaking memory is a good leaking memory example to try out the developed approach in this work. To evaluate the approach as good as possible, it will be assumed that we do not know that the Commons HttpClient library has a leaking memory that is caused by a `HashMap` not correctly deleting the elements.

Test Results on Commons HttpClient Library

First, the jar file with the small example driver and the Commons HttpClient library is woven and executed to generate the trace file and symbols file.



Choose Start Time and End Time:

0

156625

Figure 25: Generated line chart from the trace file and symbols file created through the small example driver and the Commons HttpClient library.

Afterwards, the two files are loaded and the line chart in Figure 25 is generated. Figure 25 shows that the total elements in the collections are steadily growing over time. We know from the small example driver that we execute HTTP requests multiple times where all collections used should be deleted each time. Therefore, there should be no growth in elements between each request. Analyzing the visualized data, we see that there have been eight batches and each batch increased the kept total element amount. Now we start the search for the problem of the continuous growth. We choose a time frame where the interesting part, the continuous growth, is completely included. Here, the time frame is almost the whole traced execution time, except the last part(see ① and ② in Figure 25).

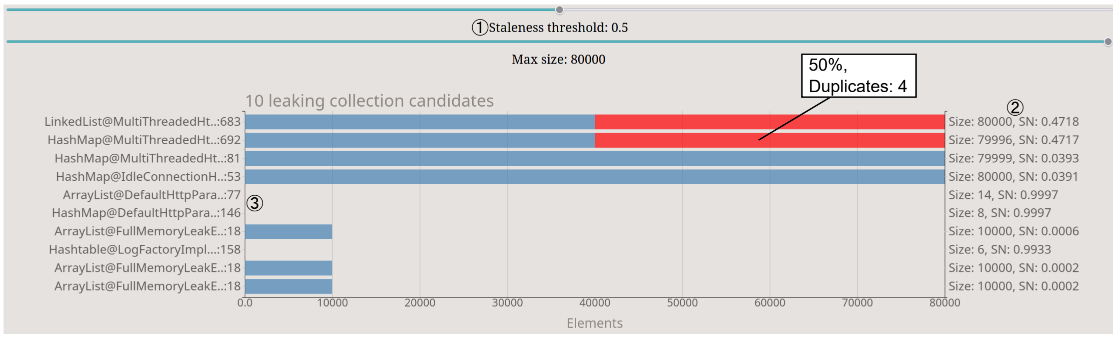


Figure 26: Staleness chart over the continuous growing elements.

The resulting staleness chart, shown in Figure 26, lists the ten most likely leaking candidates. When viewing the staleness threshold ① of 0.5, the first two collections as well as the tiny collections in the middle ③, that have too few elements to be seen, are red. But the tiny collections are in files starting with the names 'DefaultParameter' and 'Log'. That probably means that they will only be queried when logging happens or if a default parameter is requested. In both cases the small example driver is probably too specific to use them and thus we exclude them from the initial search. Far more interesting are the top two candidates. The red bar visualizes a staleness of 50 percent. When we check the size and staleness value(SN) on the right side ②, we see that they are both around 80.000 elements. When hovering over them, we see that the collection smaller than 80.000 elements has the problem of duplicate IDs for elements but is in fact also the size of 80.000 elements. They are both extremely stale with a SN of around 0.47. Hence, they are the most interesting candidates for closer inspection.

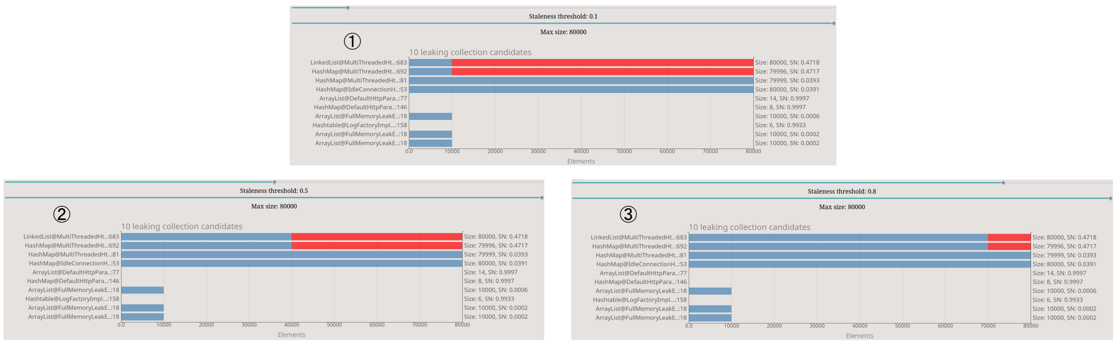


Figure 27: Three different staleness thresholds tested on the staleness chart.

First, lets check how the elements added to them change with a different staleness threshold. In Figure 27 we test the three different staleness thresholds 0.1, 0.5 and 0.8. With the smallest staleness threshold, we see that only 10.000 elements of the first two collections have a lower staleness value. When we increase the staleness threshold to 0.5, suddenly only half the elements in both collections are above the threshold. When checking the last staleness threshold of 0.8, we have only the last 10.000 elements over the threshold. From this we conclude that both collections grow continuously over time without regular accesses, which correlates with the data seen in Figure 25.

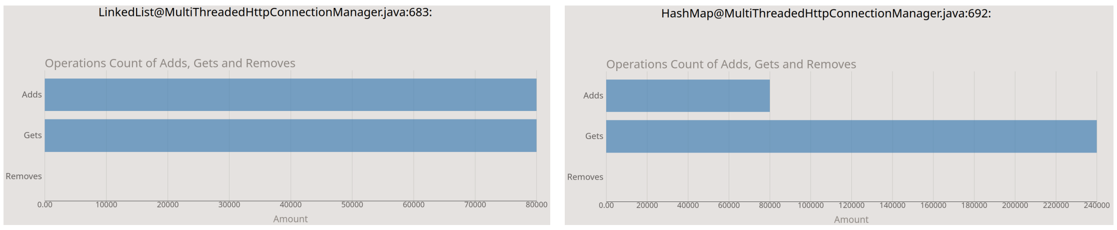
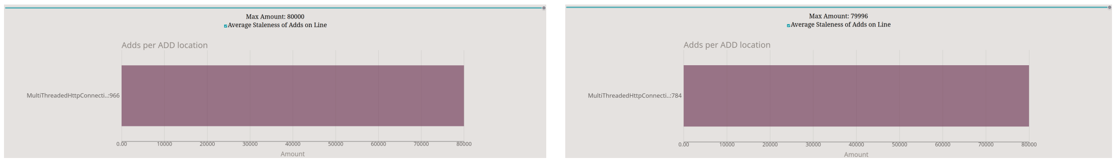


Figure 28: The operation charts for the two most suspicious collections.

Now we take a closer look at these two suspicious collections. Figure 28 shows the operation charts for both collections. The `LinkedList` on the left has the exact same amount of accesses as the amount of added elements. The `HashMap` on the right has three times the amount of accesses to the amount of added elements. Suspiciously both collections have no removes at all. When combining the knowledge of no REMOVEs and the staleness distribution seen in Figure 27, it seems that it was forgotten to delete the elements after each batch. To get a lead where most likely the removing of the elements in the source code files was forgotten, we check the distribution of the ADD locations. Figure 29 illustrates on the left (a) the `LinkedList` and on the right (b) the `HashMap`. Both are adding elements only in one location in the same file as their ALLOCATION. That gives a strong hint to start the search from there.



(a) LinkedList@MultiThreadedHttpConnectionManager:683

(b) HashMap@MultiThreadedHttpConnectionManager:683

Figure 29: The ADD locations for the (a) `LinkedList` and for the (b) `HashMap`.

For this particular problem the charts displaying the amount of accesses and the GET locations and REMOVE locations were not really useful. That was because the operation chart already showed us that both collections had no REMOVEs at all for all batches.

Concluding, we have two leaking collection suspects. One of them is a `HashMap`, which is the same collection type that was also found with the approach of Weninger et al. [27]. In conclusion, the approach of this work performed really good for this type of problem. The staleness threshold was really useful by identifying how many extremely stale elements existed in a collection. Therefore, this approach allows to find out when a collection has only few elements with a high staleness value that increases the staleness value significantly or when the elements are rather similarly distributed, as seen in this case. When there are multiple ADD locations, the staleness distribution per ADD location has a large potential to help restrict the search space on the source code level even further.

9 Related Work

There is a lot of prior work for detecting memory leaks and some works defined staleness for better detection.

First, the approach of Michael D. Bond et al. [2, 3, 4] tracks each objects staleness, i.e., the time span since the program last used a object, through a stale counter. Second, the approach taken by Trishul M. Chilimbi et al. [9] saves information for each heap object and guesses through different staleness predicates (e.g. *Never Accessed*, i.e., every object that has never been accessed is considered a leak) whether the object is leaking or not. Finally, Yan Tang et al. [24] based the staleness part of their approach on a modified version of Michael D. Bond et al. [2, 3, 4]. All three approaches are focused on tracking the staleness for each object and thus suffer from various problems (e.g. scalability, reflection, etc.) as described by Xu et. al [30].

The approach of Xu et al. [29] focuses on *container profiling*, i.e., the tracking of interactions with data structures instead of tracking the interactions with single objects. To achieve this tracking for each different and tracked data structure (e.g. in Java `Collection` classes and `Map` classes have a different method name for adding elements) a *wrapper* class, also called "glue class", is created. The purpose of the wrapper classes is to simplify the different methods of each data structure to only the operations ADD, GET and REMOVE. But for the wrapper classes to be able to simplify the methods, they need to be introduced to the source code of the monitored application. Because of this, the operations on the contained objects of the tracked data structures can be tracked. The authors choose to focus on data structures as they assumed that the misuse of them were one of the major reasons for memory leak and also led to systemic bloat.

The approach of this work was based on the work of Xu et al. [30]. But instead of creating wrappers that have to be used in the tracked applications source code, the concept of AOP was used to track selected interactions with the data structures.

10 Limitations and Future Work

The outcome of this master thesis suffers from a few limitations, but the approach still offers room for future improvements.

10.1 Limitations

There were three main limitations encountered in this work:

1. *Memory tracking*: In Java, the memory consumption of collections can only be approximately measured without using Java agents. This measuring adds extra time to the program execution for each time a collection is measured. This works fine if there are only a few collections and all collections are small. But if there is a huge amount of collections and/or very large collections, then the extra time added to the program execution for each time measuring the collections becomes very large. Depending on how many collections there are and how large these collections are, the extra time added could be far larger than the program execution of the original program. After encountering this problem without finding a solution to improve the program execution time while measuring, the measuring for memory consumption was removed entirely.

2. *Unique identifier*: Without having a unique identifier in all objects added using Java agents, the best solution found was using the method `System.identityHashCode(Object x)` to calculate an identifier for all collections and for each element in these collections. Using this method has the detriment of not getting a unique identifier every time due to overlapping hashcodes, which led to a more severe problem of duplicate identifiers and reinserted identifiers than should normally be encountered. Another solution that would provide a unique identifier would be tracking the memory changes for each object and all collections every GC run. But this solution also suffers, because tracking the elements of many collections and/or large collections adds a significant amount of time to the program execution.

3. *Parser heap memory*: The parser used in this work holds large parts of the data read from the trace file and symbols file in the heap memory. As more and more program functions were added, more and more data was held in the heap memory to provide the data on request. This led to the problem that for programs, that generate a larger trace file and/or a larger symbols file, the program would run out of heap memory space.

The first two limitations were encountered through restrictions of the programming language Java in which the tracked applications are written. When this approach is used on applications written in a different programming language, this limitations may not apply. But this also depends on how the programming language would handle memory access and if it is possible to get unique identifiers without a large performance hit.

The third limitation was a wrong design decision of this work. Instead of having that much data in the heap memory, all the data from the trace file and symbols file should have been saved into a database. The database then provides all the data needed for calculations or the data requested by the frontend. Yet, also this approach has a downside, namely poorer performance due to the required database requests.

10.2 Future Work

Due to time constraints, certain design decisions taken in this work could not be rectified. Also, there are features that could not be implemented as efficiently as we wished for.

Database The first improvement would be to correct the design decision taken in the parser and completely redesign the whole parser to use a database instead. This would allow saving data that was discarded in the current parser in favour of saving as much heap memory as possible. For example, we could store the type for each element in the collections, information we currently discard. But this has also has demerits such as the database taking up a lot of disk space and the requests for the database are slower then to the heap.

Better Element Access Tracking Saving all access locations for every element in each collection takes a huge amount of memory. Thus, only the last access location for each element was saved. Through the database it would be possible to save each accessed location in order. This would allow identifying for each GET per GET location which elements were accessed in this location. Also, it would be possible to group the elements per access time for each GET per GET location.

Element Type In Java, classes can inherit from other classes or implement interfaces. A collection in Java allows to define the type to save all elements that inherited from a specified class or implemented a certain interface. Therefore, it is possible that there are different specific types of elements in the Java collections. A chart that shows when there are different types of elements in a collection and how the different element types are distributed would help the user to understand the viewed collection in more depth. Even though this is not interesting for memory leak detection, it would be interesting for program comprehension [29, 8], i.e. the active acquiring of knowledge about a software system.

Searching When searching through the staleness bar charts for a specified collection type or to display only collections of a specified file would improve the frontend enormously. This searching ability could be expanded to different charts such as the ADDs per ADD location bar chart, GETs per GET location bar char and REMOVEs per REMOVE location bar chart to display locations from certain files.

Anti-Patterns As already mentioned in Section 7.2, only two anti-patterns were completely implemented. But there are more anti-patterns that hold potential to be used. Consequently, more of this anti-patterns should be checked for the possibility to be implemented in a meaningful way. In addition, it should also be checked if it is possible to improve the already implemented anti-patterns.

Frontend The frontend should be improved to be more user-friendly. Moreover, some features were implemented in a certain way in the beginning when it seemed to be a good idea. These features should be rewritten in a more user-friendly way.

Different Programming Languages The approach of AOP for collecting data should be tested with different programming languages and then compared how well it worked in comparison to Java. Some interesting candidate programming languages that have an AOP implementation would be Python (for example aopy [11]), C/C++ (for example AspectC++ [28]) and JavaScript (for example AspectJS [6]).

11 Conclusions

This thesis explored if the collection data gathered through the approach of AOP could deliver similar results in detecting memory leaking collections in Java to existing approaches with modifying classes and instrumenting the code with Java agents. To achieve this, aspects and Java classes were introduced to record the ALLOCATION events and DEALLOCATION events of collections, as well as converting a specified subset of collection methods to ADD events, GET events and REMOVE events.

After explaining how the data was recorded, the parser was introduced. The parser reads the data and saves the accesses per collection. The parser communicates with a graphical frontend and when requested, calculates the staleness for a specified time frame and processes the saved data.

The frontend sends requests to the parser through the users input and displays the received data. The frontend allows the user to change the time frame, which influences the data shown by most of the generated charts. For the chosen time frame a chart with the most likely candidates for leaking memory is displayed. Every candidate can be checked more closely to see the operations that occurred for it. Through the adjustable sizes for displayed candidate collections, anti-patterns for memory inefficiency can be identified. When there are irregularities, either by the staleness of the candidates or by the operations, then the source code locations for the ALLOCATION and the locations of the operations help the user to investigate the problem on the source code level.

There were parts of the approach with AOP that worked really well and other parts that did not work at all. For example, a possible memory leaking problem in the evaluation was identified rather fast and the information provided would help to directly investigate on the source code level if there is really a problem. This shows the potential of our approach. It is especially useful to inspect the behavior of the collections. Thus, there is the potential that the approach also may identify different problems beside memory leaks or that it may be used in program comprehension.

List of Tables

List of Figures

Listings

References

[1] D. Beronic, N. Novosel, B. Mihaljevic, and A. Radovan. Assessing contemporary automated memory management in java - garbage first, shenandoah, and Z garbage collectors comparison. In N. Vrcek, M. Koricic, V. Gradisnik, K. Skala, Z. Car, M. Cicin-Sain, S. Babic, V. Sruk, D. Skvorc, A. Jovic, S. Gros, B. Vrdoljak, M. Mauher, E. Tijan, T. Katulic, J. Petrovic, T. G. Grbac, and B. Kusen, editors, *45th Jubilee International Convention on Information, Communication and Electronic Technology, MIPRO 2022, Opatija, Croatia, May 23-27, 2022*, pages 1495–1500. IEEE, 2022.

[2] 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 2006, San Jose, CA, USA, October 21-25, 2006*, pages 61–72. ACM, 2006.

[3] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In G. E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[4] M. D. Bond and K. S. McKinley. Leak pruning. In M. L. Soffa and M. J. Irwin, editors, *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2009, Washington, DC, USA, March 7-11, 2009*, pages 277–288. ACM, 2009.

[5] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of Memory Inefficiency. In M. Mezini, editor, *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*, volume 6813 of *Lecture Notes in Computer Science*, pages 383–407. Springer, 2011.

[6] Dodeca Technologies Ltd. AspectJS. `https://www.aspectjs.com/`, 2023. [Online; accessed 14-November-2023].

[7] T. Domínguez-Bolaño, O. C. Fernández, V. Barral, C. J. Escudero, and J. A. García-Naya. An overview of IoT architectures, technologies, and existing open-source projects. *Internet Things*, 20:100626, 2022.

[8] W. Fenske, J. Krüger, M. Kanyshkova, and S. Schulze. #ifdef directives and program comprehension: The dilemma between correctness and preference. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 255–266. IEEE, 2020.

[9] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In S. Mukherjee and K. S. McKinley, editors, *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*, pages 156–164. ACM, 2004.

[10] R. Laddad. *Aspectj in action: enterprise AOP with spring applications*. Simon and Schuster, 2009.

[11] M. Matusiak. aopy. `https://github.com/numerodix/aopy`, 2023. [Online; accessed 14-November-2023].

[12] Mike Bostock and Observable, Inc. D3. `https://d3js.org/what-is-d3`, 2023. [Online; accessed 14-November-2023].

[13] OpenJS Foundation. NodeJS. `https://nodejs.org/en`, 2023. [Online; accessed 14-November-2023].

[14] Oracle and/or its affiliates. Java ArrayList. `https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`, 2023. [Online; accessed 14-November-2023].

[15] Oracle and/or its affiliates. Java Collection Interface. `https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html`, 2023. [Online; accessed 14-November-2023].

[16] Oracle and/or its affiliates. Java ConcurrentHashMap. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html`, 2023. [Online; accessed 14-November-2023].

[17] Oracle and/or its affiliates. Java HashMap. `https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html`, 2023. [Online; accessed 14-November-2023].

[18] Oracle and/or its affiliates. Java HashSet. `https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html`, 2023. [Online; accessed 14-November-2023].

[19] Oracle and/or its affiliates. Java Map Interface. `https://docs.oracle.com/javase/8/docs/api/java/util/Map.html`, 2023. [Online; accessed 14-November-2023].

[20] Oracle and/or its affiliates. Java Primitive Types. `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`, 2023. [Online; accessed 14-November-2023].

[21] Oracle and/or its affiliates. Java System.identityHashCode. `https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#identityHashCode-java.lang.Object-`, 2023. [Online; accessed 14-November-2023].

[22] Oracle and/or its affiliates. Java Virtual Machine Specification. `https://docs.oracle.com/javase/specs/jvms/se20/html/jvms-1.html`, 2023. [Online; accessed 14-November-2023].

[23] Oracle and/or its affiliates. Java WeakReference. `https://docs.oracle.com/javase/8/docs/api/java/lang/ref/WeakReference.html`, 2023. [Online; accessed 14-November-2023].

[24] Y. Tang, Q. Gao, and F. Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. In R. Isaacs and Y. Zhou, editors, *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings*, pages 307–320. USENIX Association, 2008.

[25] Tarnum Java SRL. Java WeakReference. `https://www.baeldung.com/java-weak-reference`, 2023. [Online; accessed 14-November-2023].

[26] A. Villazón, W. Binder, and P. Moret. Aspect weaving in standard Java class libraries. In L. Veiga, V. Amaral, R. N. Horspool, and G. Cabri, editors, *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java, PPPJ 2008, Modena, Italy, September 9-11, 2008*, volume 347 of *ACM International Conference Proceeding Series*, pages 159–167. ACM, 2008.

[27] M. Weninger, L. Makor, and H. Mössenböck. Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor. In *Working Conference on Software Visualization, VISSOFT 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 110–121. IEEE, 2020.

[28] Xerox Corporation. AspectC++. `https://aspectc.org/`, 2023. [Online; accessed 14-November-2023].

[29] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Trans. Software Eng.*, 44(10):951–976, 2018.

[30] G. 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.