Memory Leak Visualization using Evolving Software Cities

Markus Weninger[⊗], Lukas Makor[△], Hanspeter Mössenböck[⊗]
[⊗] Institute for System Software, Johannes Kepler University Linz, Austria
[△] Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

Abstract

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

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

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

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

1 Introduction

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

Most state-of-the-art memory monitoring tools do not use graphical means to visualize such a growth. Instead, they just take two heap snapshots, calculate the difference of the number of objects for every type, and display these difference values in a table. As it has been shown in other domains such as software evolution and program comprehension [8], we think that users can also profit from software visualizations in the domain of memory monitoring.

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

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

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

2 Data

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

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

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

3 Heap State Visualization

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

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

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

Many tools also show other advanced metrics beside the number of objects and use features such as color encoding to highlight certain object groups. This can easily become overwhelming and hard to interpret for novice users. Thus, we present an approach to visualize a heap state as a software city.

3.1 Buildings and Districts

In the software city metaphor, artifacts are visualized as buildings that are arranged in districts, which can again be contained in other districts. In our case, buildings represent leaf nodes of a grouping tree, while inner tree nodes are represented as districts.

Districts are flat structures. Their area is sized to enclose all their contained districts and buildings. A building is a structure with a height and an area that depends on the number of objects its tree node represents. One of our goals was to achieve building sizes that represent more-or-less realistic building measures of real-world buildings. As preliminary formulas, we came up with $2 * \sqrt[4]{n_{objects}}$ units as the height and $\sqrt{n_{objects}}$ square units as the area for buildings. Mapping units to meters, the $70,000$ objects of `Type A` from Table 1 would, for example, be represented as a building that has an area of about 264.5 square meters and a height of 32.5 meters. With adjusted formulas for height and area, the same approach could be used to visualize the city based on the number of bytes. Mixing metrics, such as using the number of objects for the area and the number of bytes for the height, is up to future work. For example, having very few very large arrays, this could result in extremely narrow building that are extremely tall if implemented carelessly, which would distort a realistic city feeling.

3.2 Layout

To layout the districts and buildings, we used the *squarified tree map* algorithm by Bruls et al. [2]. As explained in the previous section, every building has an area based on the number of objects it represents. The squarified tree map algorithm tries to shape the area of each building as an approximate square, such that they can be laid out in a way that makes the their districts again resemble squares.

4 Evolution Visualization

To visualize the memory evolution over a selected time window, we apply time traveling. According to Wettel and Lanza [6], time traveling is achieved by stepping back and forth through the history of a system while the city updates itself to reflect its current state. In our case, the history is the sequence of grouping trees generated at every reconstructed heap state in the selected time window.

4.1 Layout

It is *not* enough to visualize these grouping trees one after another. For example, buildings could be added or removed between two heap states. This would change the layout of districts and thus the position of buildings. This leads to the problem that users could hardly figure out if and which two buildings in two different heap states represent the same tree node.

To overcome this problem we apply *static position animation* [5], which creates a general city plan in which all buildings remain at the same position during the animation. To do so, all grouping trees are merged into a *meta tree*. Every node in this meta tree stores the maximum number of objects represented by the respective node at any time. Then, the layouting of the city happens *once* based on the values in this meta tree to reserve space for every building based on its largest possible area. Then, to visualize the heap at a certain point in time, buildings are centered in the space that has been reserved for them.

4.2 Memory Leak Investigation Mode

If a memory leak exists, typically certain objects groups grow stronger than others. This is especially the case if the objects are grouped by type. To make it easier for users to identify those object groups, i.e., buildings, that grew the most, certain buildings are shown in solid mode, while the others have reduced opacity, as shown in Figure 1.

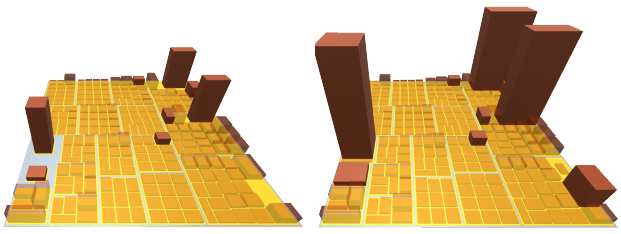

Figure 1: An application shortly after startup (left) and 60 garbage collections later (right). The ten buildings with the strongest growth are shown in solid mode, while the others have reduced opacity.

5 Interaction

Users can navigate through the city as a free-moving camera. The view can be tilted, rotated and zoomed by using the mouse wheel. By dragging the mouse or using the keyboard, the user can move the camera. Clicking on a building or district displays its information. This information includes the path from the tree root, e.g., *Overall Heap* → *Thread 1* → *Type A*, and the number of objects the structure represents.

To step back and forth in time, users are provided with buttons to go to the next and the previous heap state, as well as a slider to move through time. An automatic animation can also be played using a user-defined pause time between heap states. An example video of such an animation, showing AntTracks during trace file parsing, can be found here[1].

6 Conclusion and Future Work

In this paper, we presented an approach to visualize memory monitoring data using the software city metaphor. We discussed how a heap state, more specifically its heap objects, can be grouped into a tree, and how such a tree can be visualized as districts and buildings in a software city. Our approach is not only suitable for a single heap state, but can also visualize the memory evolution over time by using an advanced layout algorithm. Using our approach, the memory evolution of an application can be animated as a city that evolves over time, where growing buildings hint at an accumulation of objects that could be the result of a possible memory leak.

Since this work is still in progress, many possibilities exist for future work. Our current software cities only make use of three visual properties: area, height, and opacity. There is still potential for more advanced user interaction that may alter currently unused properties, such as letting users mark buildings of interest using custom colors. It is also interesting to explore how additional information such as object references could be included in a software city visualization. For example, selecting a building may also highlight / create visual links to other buildings that contain objects referenced by the selected building's objects. Also, a user study should be conducted to evaluate the usefulness of our new visualization approach.

7 Acknowledgement

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

[1] Video showing AntTracks as evolving memory city: `http://ssw.jku.at/General/Staff/Weninger/AntTracks/SSP19/MemoryCities-WorkInProgress.webm`

References

[1] C. Knight and M. Munro. "Comprehension with [in] virtual environment visualisations". In: *Int'l Workshop on Program Comprehension*. 1999.

[2] M. Bruls, K. Huizing, and J. J. van Wijk. "Squarified Treemaps". In: *Joint Eurographics and IEEE TCVG Symp. on Visualization*. 2000.

[3] C. Knight and M. Munro. "Virtual but visible software". In: *Conf. on Information Visualization*. 2000.

[4] R. Wettel and M. Lanza. "Visualizing Software Systems as Cities". In: *Int'l Workshop on Visualizing Software for Understanding and Analysis*. 2007.

[5] G. Langelier, H. Sahraoui, and P. Poulin. "Exploring the evolution of software quality with animated visualization". In: *Symp. on Visual Languages and Human-Centric Computing*. 2008.

[6] R. Wettel and M. Lanza. "Visual Exploration of Large-Scale System Evolution". In: *Working Conf. on Reverse Engineering*. 2008.

[7] F. Steinbrückner and C. Lewerentz. "Representing Development History in Software Cities". In: *Int'l Symp. on Software Visualization*. 2010.

[8] R. Wettel, M. Lanza, and R. Robbes. "Software systems as cities: a controlled experiment". In: *Int'l Conf. on Software Engineering*. 2011.

[9] F. Steinbrückner and C. Lewerentz. "Understanding software evolution with software cities". In: *Information Visualization* 12.2 (2013).

[10] F. Fittkau, A. Krause, and W. Hasselbring. "Exploring software cities in virtual reality". In: *Working Conf. on Software Visualization*. 2015.

[11] P. Lengauer, V. Bitto, and H. Mössenböck. "Accurate and Efficient Object Tracing for Java Applications". In: *Int'l. Conf. on Performance Engineering*. 2015.

[12] P. Lengauer et al. "Efficient Memory Traces with Full Pointer Information". In: *Int'l. Conf. on Principles and Practices of Programming on the Java Platform*. 2016.

[13] M. Weninger and H. Mössenböck. "User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring". In: *Int'l Conf. on Performance Engineering*. 2018.

[14] M. Weninger. *AntTracks*. 2019. URL: `http://mevss.jku.at/AntTracks`.

[15] M. Weninger, E. Gander, and H. Mössenböck. "Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection". In: *Int'l Conf. on Performance Engineering*. 2019.