Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study

MARKUS WENINGER, Institute for System Software, Johannes Kepler University Linz, Austria
PAUL GRÜNBACHER, Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria
ELIAS GANDER, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria
ANDREAS SCHÖRGENHUMER, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria

Memory analysis tools are essential for finding and fixing anomalies in the memory usage of software systems (e.g., memory leaks). Although numerous tools are available, hardly any empirical studies exist on their usefulness for developers in typical usage scenarios. Instead, most evaluations are limited to reporting the performance overhead. We thus conducted a study to empirically assess the usefulness of the interactive memory analysis tool AntTracks Analyzer. Specifically, we first report findings from assessing the tool using a cognitive walkthrough, guided by the Cognitive Dimensions of Notations Framework. We then present the results of a qualitative user study involving 14 subjects who used AntTracks to detect and resolve memory anomalies. We report lessons learned from the study and implications for developers of interactive memory analysis tools. We hope that our results will help researchers and developers of memory analysis tools in defining, selecting, and improving tool capabilities.

CCS Concepts: • **General and reference** → **Evaluation**; Metrics; Performance; • **Human-centered computing** → **User studies**; **Usability testing**; **Walkthrough evaluations**; **Empirical studies in HCI**; *Graphical user interfaces*; *User centered design*; • **Information systems** → Users and interactive retrieval; • **Software and its engineering** → Software performance.

Additional Key Words and Phrases: Interactive Memory Analysis Tools; Cognitive Walkthrough; Cognitive Dimensions; User Study; Usefulness; Usability; Utility; Assessment

ACM Reference Format:
Markus Weninger, Paul Grünbacher, Elias Gander, and Andreas Schörgenhumer. 2020. Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study. *Proc. ACM Hum.-Comput. Interact.* 4, EICS, Article 75 (June 2020), 37 pages. https://doi.org/10.1145/3394977

1 INTRODUCTION

Interactive memory analysis tools collect, process, transform, and visualize information about the memory footprint of software systems. Snapshot-based tools analyze a single point in time while trace-based tools allow users to explore a period of time [105]. For example, existing tools typically present the heap state of an application as a type histogram displaying the number of objects and

Authors' addresses: Markus Weninger, markus.weninger@jku.at, Institute for System Software, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Paul Grünbacher, paul.gruenbacher@jku.at, Institute for Software Systems Engineering, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Elias Gander, elias.gander@jku.at, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria; Andreas Schörgenhumer, andreas.schoergenhumer@jku.at, Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Altenberger Straße 69, Linz, 4040, Austria.

bytes allocated for each type. Analyzing such information allows users to detect potential memory anomalies and to reveal their root cause.

Existing interactive memory analysis tools provide a variety of capabilities to analyze different aspects of memory usage. For example, the Eclipse Memory Analyzer (MAT) [106] and VisualVM [108] are the most commonly used open source memory analysis tools for Java. While MAT purely focuses on memory analysis, VisualVM is a more general performance analysis tool including advanced memory analysis features. Kieker [58, 110, 111] is a well-known general performance framework for monitoring and analyzing the run-time behavior (including memory) of concurrent or distributed software systems. Well-known examples of commercial tools providing memory analysis features are the Dynatrace application performance monitoring (APM) platform [29] and the JProfiler [30], which offers memory profiling and a heap walker for Java applications.

So far, most memory analysis tools have been evaluated with a focus on their performance overhead and feasibility while only little empirical research exists on their *usefulness* in practical environments and for realistic usage scenarios. The term usefulness captures a tool's utility, i.e., to what degree it allows users to achieve their goals, and its usability, i.e., how well users can make use of the offered features. The study by Zaman et al. [128] is an exception in the field of performance engineering, as the authors show for two enterprise systems that test-based performance analyses need to be complemented with user-centric assessments to better understand user-perceived quality. The authors strongly argue that performance engineering should use the knowledge on how to conduct user-centric analysis from other fields.

This paper thus reports findings of a cognitive walkthrough to assess the usability of an interactive memory analysis tool. We also conducted a qualitative study to analyze the behavior of users analyzing memory anomalies in a realistic context. We performed our research using *AntTracks*, a memory monitoring system which comprises the AntTracks VM [64–66], a custom virtual machine based on the Java Hotspot VM [107], and the AntTracks Analyzer [7, 114–117, 119, 120, 122], a trace-based memory analysis tool. The AntTracks VM records memory events such as object allocations and object movements during garbage collection (GC) by writing them into trace files [64–66]. The AntTracks Analyzer then parses trace files by incrementally processing these events, thereby allowing to reconstruct the heap state for every GC point [7]. Various memory analyses can be performed with AntTracks, including heap state analysis [114, 119, 122], data structure growth analysis [115, 117], and heap evolution visualization [120].

Specifically, the contributions of our work encompass (1) a discussion of common memory analysis activities and tool capabilities based on existing research and tools (Section 3), (2) a realization of these capabilities in the AntTracks Analyzer memory analysis tool (Section 4), including an assessment based on a cognitive walkthrough following the Cognitive Dimensions (CD) of Notations Framework (Section 5), (3) the design (Section 6) and results (Section 7) of a usefulness study involving 14 participants who used AntTracks in two realistic analysis scenarios, and (4) general recommendations for researchers and developers of interactive memory analysis tools we derived from lessons learned during the study (Section 8) as well as a discussion on how we used these recommendations to further improve AntTracks (Section 9). Section 10 discusses threats to validity and Section 11 concludes the paper.

2 RESEARCH METHOD

The field of human-computer interaction (HCI) distinguishes inspection-based and test-based approaches [47] to evaluate the usability of software systems. Inspection-based techniques aim at assessing and improving interactive systems by checking them against some standard, such as Nielsen's usability attributes [80] or the Cognitive Dimensions (CD) of Notations Framework [8, 9, 11, 39–41]. Test-based techniques, on the other hand, involve end users in the evaluation.

Memory analysis is a highly complex and interactive process. Our research method thus relies on both inspections and testing. Specifically, we investigated two research questions on the usefulness of interactive memory analysis tools using the example of AntTracks: (RQ1) Regarding *usability* we assessed AntTracks' memory analysis capabilities from the perspective of software engineers, guided by the CD framework and Nielsen's usability attributes. (RQ2) Regarding *utility* we conducted a user study analyzing a real-world Java web application with seeded memory defects. Based on these results, we synthesized recommendations and lessons learned intended to support developers of interactive memory analysis tools. To tackle these questions, we conducted our research in four steps:

Identification of Memory Analysis Activities. We studied related research and features of state-of-the-art memory analysis tools to identify important memory analysis activities benefiting from tool support. In addition to that, we present how these memory analysis activities manifest themselves in the memory monitoring tool AntTracks, the main subject of this study.

Cognitive Walkthrough and Tool Improvement. To assess AntTracks' usability, we first performed a cognitive walkthrough of the identified activities using the CD framework, which offers a vocabulary for discussing usability issues and their trade-offs. The CD framework has been used successfully to assess software tools [5, 60, 70, 90, 91], visual diagrams [10], temporal specification notations [62, 63], or visual modeling languages [22, 123]. Table 1 shows a summary of these dimensions. A detailed description of the framework and the cognitive dimensions can be found online [40]. The primary aim of this cognitive walkthrough was to reveal and fix possible usability flaws before conducting the user study and to define the scope for the user study.

User Study. We designed our study based on the findings from the CD assessment and the guidelines for conducting empirical studies by Runeson and Höst [97]. Software engineering students from our university used AntTracks to investigate the memory evolution of an application to detect anomalies such as memory leaks or high memory churn (cf. Section 6). For each anomaly, the participants aimed at revealing its root cause using the memory analysis. During this process, we asked each study participant to 'think aloud' [47], i.e., to describe what they were doing and to

Table 1. Cognitive dimensions used for the walkthrough (taken from [40]).

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

comment on any concerns. The participants were interviewed on the utility of the tool [23] and also completed a usability questionnaire [80].

Derivation of Implications. Finally, we synthesized recommendations and lessons learned based on the detailed results and feedback obtained from the study. In addition, we discuss how these recommendations were used to further improve AntTracks.

3 MEMORY ANALYSIS ACTIVITIES

We present key activities supported by interactive memory analysis tools based on our experiences and related work. We focus on memory analysis for managed languages such as Java or C#. We will show that the tools vary regarding their support, e.g., some tools only visualize raw data and leave the analysis to the users, while other tools automate certain analyses activities.

3.1 Collecting Memory Data

Basic tools for snapshot-based inspections of memory usage mostly rely on heap dumps, which can be created by tools such as HPROF [85, 88] or jmap [84]. The following techniques are used for analyzing more specific details of snapshots or memory usage over time [6]: (1) A *modified execution environment* such as a custom Java VM that can access internal information; (2) a *sampling-based approach*, e.g., an agent using the Java VM Tool Interface [87] to receive periodical callbacks about memory-relevant events in the application; or (3) an *instrumentation-based approach* that relies on adding code to an existing application, either before compilation (e.g., AspectJ [57]) or at run time (e.g., ASM [14, 15, 61] or Javassist [17, 18]).

3.2 Detection of Memory Anomalies

Before inspecting an application in detail, memory analysis tools support users in detecting memory anomalies such as memory leaks, high memory churn, memory bloat, or unusual GC behavior.

3.2.1 Memory Leaks.

Memory leaks [35] in managed languages occur if objects no longer needed remain reachable from garbage collection roots (e.g., static fields or local variables) due to programming errors. For example, objects may accumulate over time when a developer forgets to remove them from long-living data structures [115]. Such leaks lead to a growing memory footprint, which at some point will cause an application to crash. There are two main approaches to detect memory leaks: (1) Techniques detecting *staleness* [12, 44, 92, 126] assume that objects not used for a long time are likely involved in a memory leak. However, the proposed techniques are hardly used outside academia due to their high costs of tracking objects. (2) Techniques detecting *growth* [16, 54, 76, 104] are thus still the de-facto standard in state-of-the-art memory analysis tools and mostly rely on users interpreting visualizations. For example, VisualVM [108] periodically plots the memory footprint in a time-series chart. The user then has to check for suspicious sections of continuous growth that might hint at a memory leak. Similarly, JConsole [83] can read a running application's Java Management Beans to plot the currently occupied heap memory separated by eden space, survivor space, and old space.

3.2.2 Memory Churn.

Memory churn occurs when large numbers of short-living objects are created by an application, thereby causing many garbage collections. Such excessive dynamic allocation behavior [102] typically has a negative impact on performance. However, obtaining the information on how long objects survive before dying is expensive [45, 95, 96]. Most tools are thus limited to analyzing the number of allocations, but not the exact lifetime of objects. Objects frequently allocated in burst typically do not survive for a long time and thus the high allocation rate already indicates memory

churn. Memory churns can be detected either by visually spotting spike patterns in memory charts (i.e., high consumption of memory followed by many object deaths) or by plotting the number of allocations over time (i.e., detecting allocation-intensive time windows), as for example done in Dynatrace [29] or Kieker [58, 110, 111].

3.2.3 *Memory Bloat.*

Memory bloat [52, 75, 125] describes the inefficient use of memory for achieving seemingly simple tasks. It is often caused by heavily using (object-oriented) abstractions, such as in over-generalized data structures. Most techniques for detecting memory bloat thus focus on analyzing data structures requiring many auxiliary objects [77] or inefficient usage of data structures operations for adding, getting, or removing elements [124, 127].

3.2.4 *Unusual GC Behavior.*

The behavior of the GC can also indicate memory problems. Instead of looking at the memory behavior of an application, this anomaly is detected by inspecting the garbage collector, e.g., by measuring GC overhead via the garbage collection count and the garbage collection duration.

3.3 Inspection of Memory Anomalies

Once a suspicious memory behavior is detected, the user can inspect a single point in time or a time interval to reveal the root cause of the problem.

3.3.1 *Single Point in Time.*

The most common technique is a *heap state analysis*, which relies on reconstructing the objects that were alive at a certain point in time. For every object on the heap, a number of properties can be reconstructed depending on the tool: these may include the object's address, its type, its allocation site, the heap objects it references, the heap objects it is referenced by, the thread allocating the object, and a list of root pointers referencing it. Users can then examine (groups of) objects on the heap or study metrics about the heap state. *Object-based techniques* allow to inspect heap objects in a bottom-up or top-down fashion [114]. In the bottom-up approach the user searches for big object groups (e.g., objects of the same type) and then tries to free them. The most common visualization to find these object groups is a type histogram grouping all heap objects by their types, and also showing the memory occupied by each type. The object type(s) consuming most memory can then be inspected in detail. Some tools support users by displaying the path to the GC roots, while other approaches assist users by displaying the code that has allocated the objects. Visualization approaches [2, 46, 71, 74, 93, 98, 129] aggregating the object graph (e.g., based on its dominator tree [67, 73, 114]) are useful to analyze the heap's composition. A user following the top-down approach first selects a GC root or a heap object that keeps alive many other objects. The user then inspects the objects reachable from this root or object and searches for possible cut points in the path [114, 115]. *Metric-based techniques* derive metrics from the heap state that allow to analyze the heap state by revealing fields, objects, classes or packages that are likely involved in memory anomalies [19, 20, 77].

3.3.2 *Evolution over Time.*

A number of tools also allow to analyze the memory usage evolution of an application over time [24, 25, 76, 116], in its basic form by comparing heap states. In MAT [106], for example, users can compare two heap states by computing a delta type histogram diagram to identify objects with high growth rates. In Dynatrace [29] users can show the number of objects allocated in the selected time interval. Extensively allocated objects can then be considered for reuse, caching, or removal. Other approaches allow to automatically detect growing data structures [115, 117], or to visualize the evolution of the memory composition over time [120, 121].

4 OVERVIEW OF ANTTRACKS

The first result of the AntTracks project [113] was a custom Java VM for efficiently collecting detailed memory traces [65]. The AntTracks Analyzer then started as a research prototype for reconstructing heap states from these memory trace files [7]. The tool is now an interactive memory analysis tool for the detection and inspection of various memory anomalies. We selected the AntTracks tool as subject for this study as it is a publicly available[1] and covers more memory analysis tasks than alternative tools. For example, AntTracks can perform detailed analyses over time due to its trace-based nature, while other publicly available tools such as MAT [106] or VisualVM [108] are restricted to snapshot-based (dump-based) analyses. Another reason for selecting AntTracks was the high familiarity of some authors with its code base. The goal of the cognitive walktrough was to reveal and fix major flaws before the user study, so detailed knowledge of the tool and its implementation was essential.

In the following, we give an overview of a subset of AntTracks Analyzer's features, organized by the memory analysis activities presented in Section 3.

4.1 Memory Growth Detection — Overview

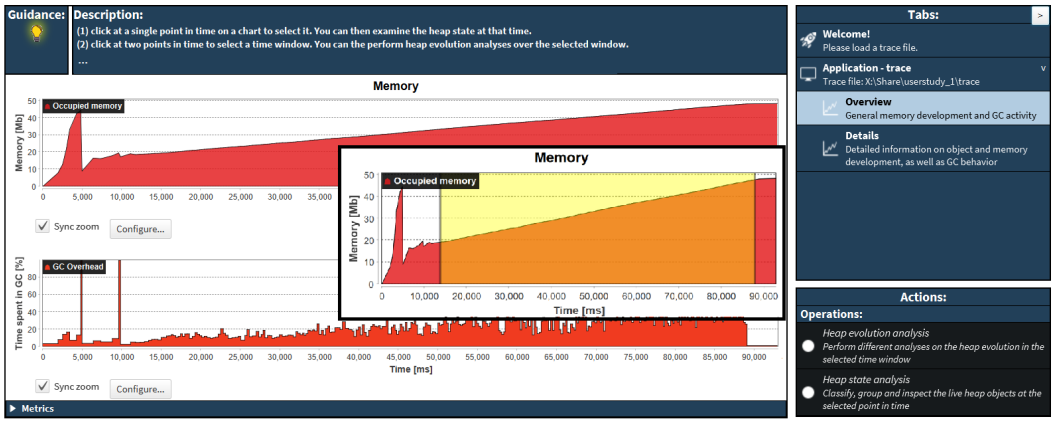
Users working with the AntTracks Analyzer first open a trace file recorded with the AntTracks custom VM. The file contains information on the memory behavior of the monitored application. The *application overview* (see Figure 1a) opens upon loading and shows the memory footprint and GC overhead as time-series charts. A continuous growth of the memory footprint, for instance, may indicate a memory leak. This overview is intentionally kept simple. For example, to avoid terminology unknown to the user, the memory footprint chart only contains a single time series showing the occupied memory. Moreover, it only shows data points marking the end of garbage collections, thereby resulting in a smoother trend line[2].

4.2 Memory Growth Inspection: Evolution over Time — TrendViz View

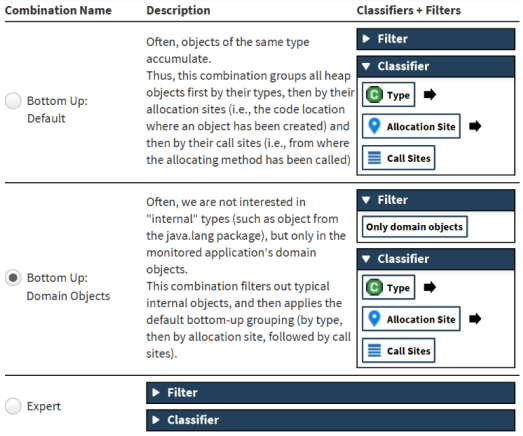
If a user detects a time window with suspicious memory growth, AntTracks' *TrendViz view* [120] allows to inspect the memory evolution during this time window in more detail. The first step is to define properties based on which the heap objects are grouped during analysis (see Figure 1b). For this purpose, AntTracks provides a variety of different object classifiers [119, 122], each of which groups heap objects based on a different criterion. For example, the type classifier groups all objects by their type name, e.g., `java.util.HashMap`. A user can select multiple classifiers for grouping the heap, which results in a classification tree. For example, using the type classifier followed by the allocation site classifier first groups all objects based on their types, and then further groups all objects of a given type based on the source code location they were allocated at. The AntTracks TrendViz visualizes the evolution of the heap based on the selected classifiers (see Figure 1c). When opening the view, a single chart shows only the evolution of the first level of the classification tree, e.g., the evolution of the objects grouped by type. The user can then display further charts for the next levels of the classification tree, e.g., the evolution of the allocation sites of a selected type. For example, in Figure 1c the most-allocated type `Product` has been selected by the user in the top chart (highlighted in yellow), and a second chart below displays this type's allocation sites. This way users can interactively collect information about suspicious objects accumulating over time.

[1] AntTracks download link: http://ssw.jku.at/General/Staff/Weninger/AntTracks/Publish/

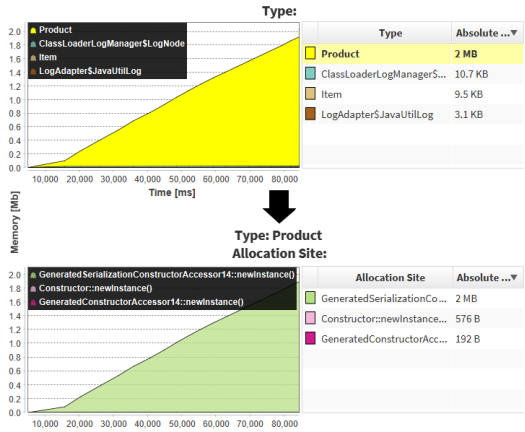
[2] The occupied memory is generally higher when a garbage collection starts, however, the spikes between garbage collection starts and ends are not relevant for the purpose of detecting memory leaks.



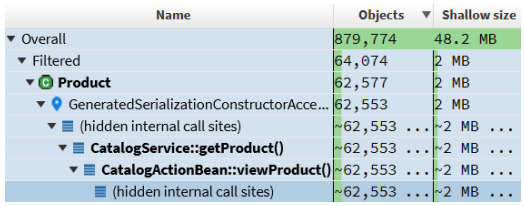
(a) The *Overview* plots the application's memory footprint and GC overhead and allows to select a suspicious memory leak time window.



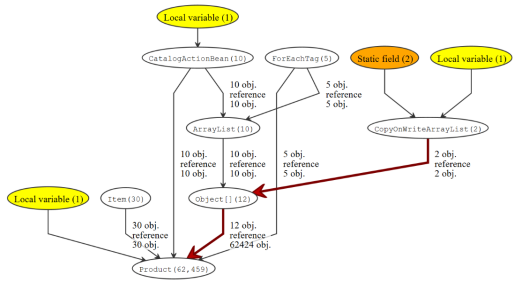
(b) Users can choose from a list of classifier combinations to group heap objects.



(c) The *TrendViz* displays the heap evolution grouped by the selected classifier combination.



(d) The *heap state view* displays the classified heap at a certain point in time as a tree table.



(e) The *graph view* highlights the path from a selected group of objects (shown at the bottom) to its most important GC roots (colored nodes).

Fig. 1. Memory leak analysis in AntTracks.

4.3 Memory Growth Inspection: Single Point in Time — Heap State View + Graph View

Users analyzing memory growth over time often reveal suspicious objects that accumulate memory. These objects can then be further inspected at a specific point in time. For example, after a memory growth analysis, AntTracks may suggest to inspect the heap state at the end of the previously selected time window. At this point, all objects that have accumulated during this time window are present in the heap and can thus be easily inspected. AntTracks can visualize the heap state using a *table-based* or *graph-based analysis*.

4.3.1 Table-based Analysis — Heap State View.

When inspecting a specific heap state, the user first selects a classifier combination (cf. Figure 1b) for grouping the heap objects. The resulting classification tree is displayed in a tree table on the *heap state view*, as shown in Figure 1d. In this table, the user can further inspect suspicious objects previously identified in the trend view. For example, this view allows to inspect the *GC closures* of an object group [114], i.e., the objects kept alive by a certain object group, or a tabular visualization of the path to the closest GC root [114], similar to VisualVM [108].

4.3.2 Graph-based Analysis — Graph View.

Further analyses are needed if a user detects a suspiciously large group of objects being kept alive. This can happen in garbage-collected languages if objects are still directly or indirectly reachable from GC roots such as local variables or static fields. In this case, the user needs to inspect the paths to these GC root to find ways for reducing the number of paths.

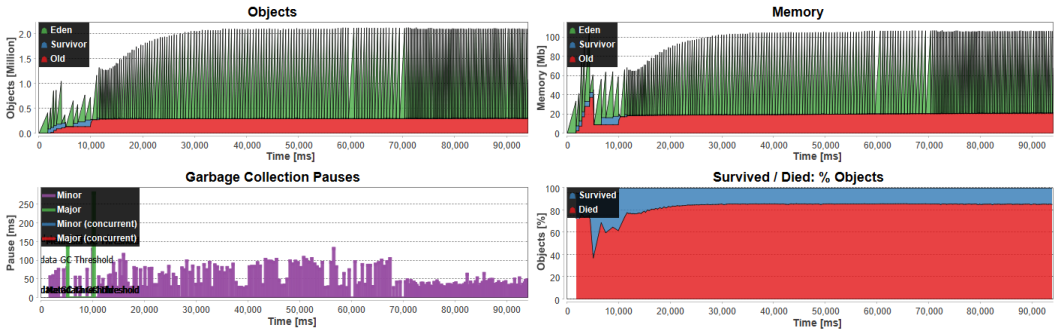
The most convenient way to inspect the paths to GC roots is the *graph view* shown in Figure 1e. Initially, this view only shows a single node representing the set of suspicious objects. By selecting the node the user can apply the *Path to GC roots* operation, which traverses the references pointing to the given objects recursively until GC roots are found. To keep the number of displayed nodes low, objects of the same type are grouped into a single node. Nodes are labeled with their objects' type name and the number of objects belonging to them. Edge labels show how many objects of the top node reference how many objects of the bottom node. GC roots are displayed as special nodes that are highlighted by a colored background. After performing the path to GC roots action, the user can explore the resulting paths and detect the GC roots referencing most objects. To make objects eligible for garbage collections, a developer can then 'cut' the paths to these GC roots by setting references to `null` or by removing objects from their containing data structures.

4.4 Memory Churn Detection — Details View

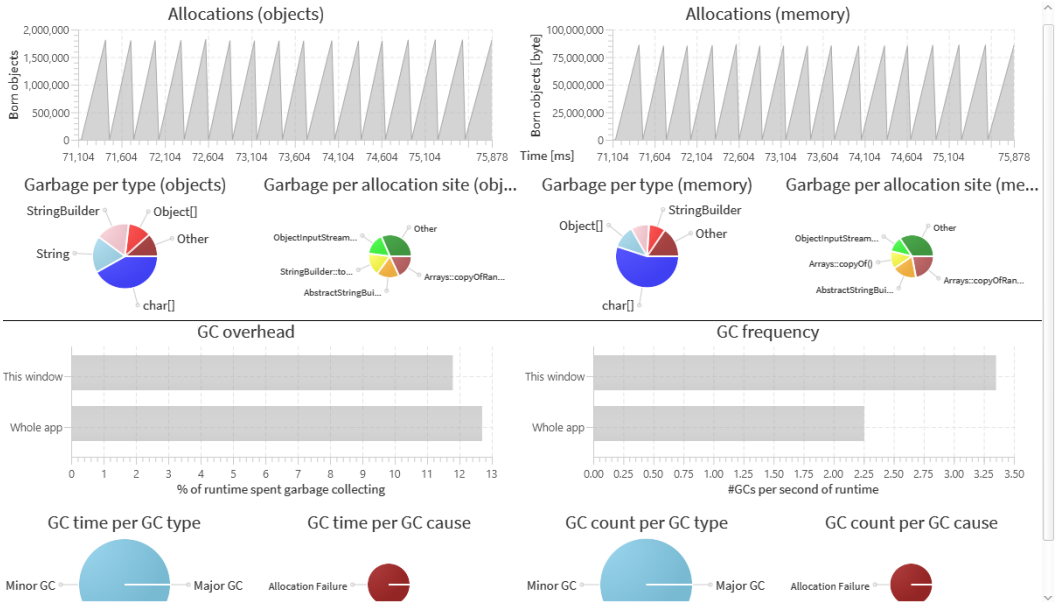
In case of *memory churn* the performance degradation is caused by the creation and garbage collection of many short-living objects [102]. In AntTracks suspicious time windows with high memory churn can be detected in the *details view*, which plots the memory footprint at the beginning and at the end of every garbage collection (cf. Figure 2a). The memory occupied at the start of a garbage collection is usually much higher than at its end, i.e., the garbage collections appear as spikes. A user aiming to detect high memory churn needs to look for high and frequent spikes in this memory footprint chart.

4.5 Memory Churn Inspection: Evolution over Time — Short-living Objects View

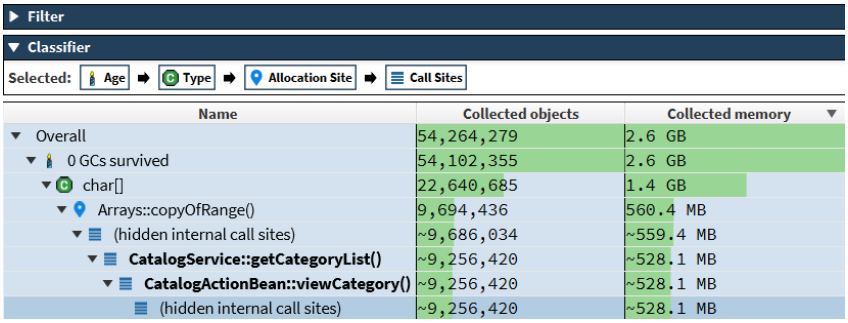
Once a suspicious memory churn time window is detected, the goal of the developer is to reduce the number of allocations by determining the objects responsible for most of the memory churn within this window. Knowing the types and allocation sites of these objects, then allows to track down their allocations in the source code to fix the problem.



(a) The *Details view* plots the application's detailed memory footprint and GC pauses and allows to select a suspicious memory churn time window.



(b) The charts on the *short-living objects view* show the monitored application's garbage collection behavior.



Filter

Classifier

Selected: Age ➡ Type ➡ Allocation Site ➡ Call Sites

Name	Collected objects	Collected memory
Overall	54,264,279	2.6 GB
0 GCs survived	54,102,355	2.6 GB
char[]	22,640,685	1.4 GB
Arrays::copyOfRange()	9,694,436	560.4 MB
(hidden internal call sites)	~9,686,034	~559.4 MB
CatalogService::getCategoryList()	~9,256,420	~528.1 MB
CatalogActionBean::viewCategory()	~9,256,420	~528.1 MB
(hidden internal call sites)	~9,256,420	~528.1 MB

(c) This table on the *short-living objects view* enables users to drill down into object groups causing suspiciously high memory churn.

Fig. 2. Memory churn analysis in AntTracks.

AntTracks detects short-living objects based on the number of garbage collections they survived. This object age information is then visualized in the *short-living objects view* comprising an overview tab presenting various garbage collection metrics to give a first impression of garbage composition (cf. Figure 2b), and an inspection tab depicting all garbage-collected objects in a tree table using AntTracks' classification mechanism (cf. Figure 2c). The overview tab helps the user to familiarize with the garbage collector activity in the selected time window. For example, pie charts reveal the object types and allocation sites producing most garbage. By selecting a chart entry the user can switch to the inspection tab to further investigate the respective objects. The inspection tab uses a tree table, similar to the one used on the heap state view, to display the garbage collected objects. Inspecting this tree allows users to determine the objects which did not survive a single garbage collection and to investigate their types, allocation sites, and the methods calling the allocating method.

5 COGNITIVE WALKTHROUGH OF ANTTRACKS

We first performed a cognitive walkthrough of the AntTracks Analyzer using the CD framework to assess its usability and to select specific usability aspects for in-depth investigation in the user study. Specifically, three authors of this paper (two of which are familiar with AntTracks' source code) independently assessed AntTracks' features for the identified memory analysis activities. To do so, every assessor performed a memory leak analysis on the *Dynatrace easyTravel* application [28] and a memory churn analysis on the *http-finagle* benchmark of the Renaissance benchmark suite [89]. Both of these applications have already been used in related work to present typical memory problems [116].

Each assessor performed the respective memory analysis task on each view of AntTracks. As cognitive dimensions are designed as 'discussion tools' [40], every assessor took notes for each view based on the 14 cognitive dimensions defined in the CD framework [8]. They also rated each cognitive dimension on each view using a color-based three-level classification: (1) green – no issues found, (2) yellow – room for improvement, and (3) red – serious flaws.

After every assessor had independently performed these tasks, their results were merged based on their ratings and their notes during a discussion session. For the rating, they always took the lowest rating of all three assessors as the joint result to ensure all concerns are addressed, i.e., if two assessors rated a CD as green and one rated it as red, the joint rating was red. Each author's notes and comments were merged and discussed to ensure a common understanding.

This merging session resulted in a single spreadsheet shown in Figure 3. The assessor comments are only partially shown due to space constraints. Six cognitive dimensions were regarded as cross-cutting, i.e., affecting the whole application. Overall 43 view-CD-pairs were rated as yellow (room for improvement) and seven cognitive dimension on two different views were rated as red (serious usability flaws). These flaws had to be fixed before the user study to prevent obvious showstoppers during the study.

Due to their high number, further evaluating all of these issues during the user study would have been infeasible. Thus, the assessors jointly selected the 27 most interesting usability issues (highlighted using black font and thick borders in Figure 3). A list of these cognitive dimensions can also be found in Table 2. This table contains one entry for every view in AntTracks alongside its respective memory analysis activity. For every view, it lists the cognitive dimensions chosen for further investigation during the user study. For each cognitive dimension, we agreed on the methods how data for evaluating the respective cognitive dimension should be collected during the user study: By *observation* (OBS) of user activities and think-aloud statements or by a specific question in the *interview* (INT) at the end of the study. The last column shows the degree of support

of a cognitive dimension based on the results of the user study. These results which will be discussed in more detail in Section 7.2.

In the following, we present the assessors' feedback to the cognitive dimensions selected for detailed inspection during the user study. The user study design as well as the interview questions have been adjusted to gain as much insight as possible into these possible usability flaws.

5.1 Memory Growth Detection — Overview

Error-proneness, i.e., the notation invites errors, has been recognized as a likely problematic cognitive dimension in the overview. The most common operations on this view are chart interactions such as the selection of a single point in time (e.g., to select the heap state to be analyzed), the selection of a time window (e.g., to select an interval for heap evolution analysis), zooming, or panning. Different

	Task	Detection: Memory Growth	Inspection: Evolution over Time	Inspection: Single Point in Time	Inspection: Single Point in Time	Detection: Memory Churn	Inspection: Evolution over Time	Cross-Cutting
	AntTracks View	Overview	TrendViz View	Heap State View	Graph View	Details View	Short-living Objects View	
Cognitive Dimensions	Abstraction	Overview uses easy terminology.	**Abstraction into chart series -> improve by ...**	Maybe terminology? Data structure DSL ...	Nodes represent groups of objects -> understandable? ...	GC chart	Is the content of the tree view clear?	**Terminology, icons, etc.**
	Closeness of Mapping	GC chart	Drill-down feature may not be clear. The hierarchical ...	Tree visualized as hierarchical TreeTableView.	How to display different elements (Objects, GC roots, ...	GC chart	Tree visualized as hierarchical TreeTableView.	
	Consistency	Evolution data is by default presented as charts in AntTracks.		Hierarchical data is by default presented as TreeTableView ...	**To achieve immersion and closeness of ...**		**Other column names than on heap state view.**	**Are there annoying inconsistencies?**
	Diffuseness		**Overcrowded classifier selection, also see viscosity.**	**Classifier selection is too complex. Highlight most ...**	**Test that not too many different no-tations are used, ...**	Explanatory text is too long.	**Many charts on overview - too many?**	**Unnecessary or unnecessarily complex views?**
	Error-proneness	**Possible flaw: Chart interaction. Positive: Zoom ...**		Opertations in context menu clear? User-defined ...	Make sure that operations that would create too ...	**See Overview (Chart interactions)**		
	Hard Mental Operations	**Do users recognize growing memory as problem?**	See abstraction & closeness of mapping.	User is free to use any classifier com-bination. Certain ...	**Even though users can inspect graphs, the detection of ...**	**Interpretation of charts hard?**	Normal classification trees.	
	Hidden Dependencies	Zoom is synced, selection is synced.	Highlight selection in parent chart better. Also display ...	**BUG: New classification in heap state may ...**			Link from pie chart to table clear?	**Are there any dependencies that we did not find yet?**
	Premature Commitment		Time window has to be selected beforehand ...	Time has to be selected beforehand	Time has to be selected beforehand. Once nodes are ...			**Order of operations, etc.**
	Progressive Evaluation	User can check how many of the suggested time ...	Selected value is shown for every level. The more levels, ...	Position withing classification tree determines progress.	User can always check the path he/she has already ...			
	Provisionality	Can open a new heap state without problems, can ...	All settings can be changed arbitrarily.	Abortion of long running operations is possible.	View is always reset-able. Future work: "What-if"-games.			
	Role-expressivness	Memory chart clear. GC chart probably not directly clear.	**Is it clear what a single chart is showing?**	**Should be clear, ask if the tree table visualization was ...**	**Are the different types of nodes clear?**	Charts maybe not clear, check if users understand what ...	**Do users understand the charts?**	
	Secondary Notation							
	Viscosity		**Inflexibility of the classifier selection. Classifiers cannot ...**	**Order of classifiers cannot be changed using drag-and-...**	Graph grows rather fast.		Order of classifiers cannot be changed using drag-and-...	
	Visibility	New overview tab was implemented: Now Memory + GC ...	**Drill-down feature has been improved (with table, etc.) ...**	Should be clear, ask if the tree table visualization was ...	Legend was needed.	**Many charts at once, may be overwhelming.**		**Tab system. Do users find out ...**

Fig. 3. The spreadsheet documents and classifies the results of the CD assessment. Each column represents an activity (cf. row 1) performed on one of AntTracks' views (cf. row 2). Each of the 14 cognitive dimensions [8] is shown in a separate row. Green cells represent cognitive dimension for which no issues were found on the respective view, yellow cells highlight cognitive dimensions on the respective view with room for improvement (possible subjects for more detailed evaluation in the user study), and red cells highlight serious problems (requiring fixes before the user study). The text in the cells shows parts of the notes taken by the inspectors during the walkthrough. Highlighted view-CD-pairs (cells with black text and thick border) have been chosen to be evaluated in more detail during the user study.

interaction mechanisms (clicking, double-clicking, dragging, etc.) for different actions exist and vary across applications, which can easily lead to misuse. *Hard mental operations* also require attention in the user study. Even though users can be expected to immediately spot continuous growth in a time-series chart, novice users might not relate such patterns to possible memory leaks that should be investigated further.

5.2 Memory Growth Inspection: Evolution over Time — TrendViz View

Diffuseness, i.e., the verbosity of the notation, and *viscosity*, i.e., the resistance to change, both needed fixing before the user study due to AntTracks' complex classifier system. Although this system is very flexible for expert to arbitrarily group heap objects, this flexibility can make the system difficult to use for novices. In particular, an overwhelmingly large list of available filters and classifiers is presented to the users (diffuseness), who may struggle to select sensible classifier combinations without a solid background in memory analysis. Additionally, the selection and arrangement of

Table 2. The cognitive dimensions that were chosen based on the results of the cognitive walkthrough to be inspected in more detail during the user study using observations (OBS) and interview questions (INT).

Activity / Capability	Tool Views	Cognitive Dimension	Assessment in User Study	Study Result (cf. Section 7.2)
Detection: Memory Growth	Overview	Error-proneness	OBS + INT	-
		Hard Mental Operations	OBS + INT	**o**
Inspection: Evolution over Time	AntTracks TrendViz View	Abstraction	OBS	**o**
		Diffuseness	OBS	**+**
		Role-Expressiveness	OBS + INT	**+**
		Viscosity	OBS	**+**
		Visibility	OBS + INT	**o**
Inspection: Single Point in Time	Heap State View	Diffuseness	OBS + INT	**+/o**
		Hidden Dependencies	OBS	**+**
		Role-Expressiveness	OBS + INT	**o**
		Viscosity	OBS	**+/o**
Inspection: Single Point in Time	Graph Visualization View	Consistency	OBS	**+**
		Diffuseness	OBS	**+**
		Hard Mental Operations	OBS + INT	-
		Role-expressiveness	OBS + INT	-
Detection: Memory Churn	Details View	Error-proneness	OBS	-
		Hard Mental Operations	OBS	-
		Visibility	OBS	**+**
Inspection: Evolution over Time	Short-living objects View	Consistency	OBS + INT	**+**
		Diffuseness	OBS	-
		Role-Expressiveness	OBS + INT	**o**
Cross-Cutting		Abstraction	OBS + INT	-
		Consistency	OBS + INT	**+**
		Diffuseness	OBS + INT	**o**
		Hidden Dependencies	OBS	**+**
		Premature Commitment	OBS + INT	**+**
		Visibility	OBS	**+**

these combinations was tedious, for example drag-and-drop features were missing (viscosity). Thus, we extended AntTracks with pre-defined classifier combinations for common tasks before the study (cf. Figure 1b). For example, the combination `Bottom-up analysis: Domain objects` first applies a filter to omit objects from internal packages (such as `java.lang` or `java.util`), and then groups the remaining objects by their types, followed by their allocation sites and by their call sites. *Visibility*, i.e., the ability to view components easily, was another showstopper CD we fixed before the user study. The view allows to select a certain object group for drill-down inspection by clicking on its chart series. Yet, the walkthrough revealed that it might not be obvious that an object group can be selected by clicking on the chart. Thus, we added a table next to the chart to make interaction abilities more visible (cf. Figure 1c) and investigated in the study whether users benefit from this additional table. *Abstraction* and *role expressiveness* question if users understand the meaning and the visualization of the drill-down process, i.e., how a classifier combination and the resulting classification tree are represented by multiple drillable subcharts displayed below each other. To emphasize the hierarchical relation between two charts, we added arrows in between charts before the study, as well as a textual description of the drill-down selection, as shown in Figure 1c. Another abstraction we considered to simplify analyses in AntTracks regards the way of presenting allocation sites and method calls. Call chains can become quite long (multiple 10s of calls) and thus hard to inspect, especially if an application employs various libraries that call each other. To reduce the amount of entries in such a call chain, AntTracks creates artificial entries labelled (`hidden internal call sites`) that combine multiple *internal call sites*, i.e., calls from one method to another inside a packages that cannot be modified by the user (such as `java.util`). This abstraction may be hard to understand for some users.

5.3 Memory Growth Inspection: Single Point in Time — Heap State View + Graph View

5.3.1 Table-based Analysis — Heap State View.

Diffuseness and *viscosity* are also relevant on this view since it uses the same classifier system as the TrendViz view discussed in the previous section. Yet, different classifier combinations are required on both views, as certain combinations are only sensible when analyzing a single heap state but not a trend. We thus improved the system by adding even more pre-defined classifier combinations, but showing only the relevant ones on the respective views. *Role expressiveness* on this view's classifier selection questions whether the different combinations can be distinguished and understood by the users. For each combination, AntTracks shows its name, a description and the list of used filters and classifiers, as shown in Figure 1b. We added an interview question to clarify if this explanation is sufficient for users. *Hidden dependencies* were another problem that became apparent during the cognitive walkthrough. Users can select an object group in a heap state view and then apply various operations, some of which open new windows displaying information related to the selected object group. If the heap state window changes, for example, by selecting a different classifier combination, the object groups on the child windows no longer exist in the parent window, thus breaking (hidden) dependencies. We tried to prevent this problem by opening a new heap state window every time a new classifier combination is applied.

5.3.2 Graph-based Analysis — Graph View.

Consistency is a concern on this view, since its graph-based visualization strongly differs from other AntTracks views, most of which use time-series charts and tree tables to display data. *Role-expressiveness* has to be evaluated regarding the different types of nodes, edges, color encodings and other features that are intended to help users to understand the graph but also pose the risk of being too complex. *Diffuseness* may also be affected by this graph notation. Grouping objects based on their type significantly reduces the number of nodes on the screen but requires additional labeling.

This could result in an overly high number of screen objects negatively affecting comprehension by the user. *Hard mental operations* have to be performed as users try to spot suspicious GC roots and suitable cutting points on the paths to them. The view can visualize the heap object graph and roots to the GC roots, yet this information is only useful if users are able to interpret it correctly.

5.4 Memory Churn Detection — Details View

Error-proneness, as in the overview, concerns the interaction with AntTracks' time-series charts, which are the main way of visualization on the details view. *Visibility* asks the question how easily users can detect time windows with high memory churn, i.e., spike-patterns with frequent and tall spikes on the chart, on the details view. *Hard mental operations* are potentially required for users without experience in memory analysis to correctly interpret such spikes as suspicious.

5.5 Memory Churn Inspection: Evolution over Time — Short-living Objects View

Role-expressiveness should be assessed during the study on AntTracks' short-living objects view. Various information (such as garbage composition) is visualized using pie charts, but not all of it may be clear to the users, e.g., due to the terminology used. *Diffuseness* is also of interest as the view contains twelve charts, some of which contain less crucial information and thus could diffuse the more important information. *Consistency* regards the tree table on the inspection tab. In other AntTracks views, the tree table shows the live objects of a certain heap state, while this tree table shows all objects garbage-collected in the chosen time window. We decided to investigate this break in consistency in the study.

5.6 Cross-Cutting Dimensions

Several cognitive dimensions were found to be relevant for all views of AntTracks. We decided to assess in the study if these are well supported. *Visibility* and preventing *hidden dependencies* were our main concerns when choosing a stacked tab arrangement in AntTracks. AntTracks offers a number of different analysis features, many of which open new (child) views. We thus decided on a stacked tab system where each tab can again have further child tabs. *Abstraction* is also important in nearly all tools. For example, typical abstractions are icons or terminology inherent to the given domain. Certain abstractions may be hard to understand in which case they should be fixed in the future. *Consistency* is important for visualizations in tools. AntTracks mostly uses the same chart style on all of its views. Also, much of the information in AntTracks is visualized in tables, often tree tables, since most of its data is arranged in trees (for example classification trees). We included a question regarding consistency in the study questionnaire to reveal potential inconsistencies. *Diffuseness* may concern especially non-expert users. Visualizations should be as clear as possible, and the study was designed to reveal unnecessary or unnecessarily complex parts of the tool.

6 USEFULNESS STUDY DESIGN

Based on the results of the cognitive walkthrough and the subsequent improvements of the tool, our qualitative study assessed the usefulness (i.e., usability and utility) of AntTracks' memory analysis capabilities. We structured our study using the guidelines by Runeson and Höst [97]. For the design of the study tasks, we followed the recommendations by Ko et al. [59]. Specifically, we defined six tasks based on the the initial survey of memory analysis activities and capabilities (cf. Section 3). We iteratively refined these tasks by first testing their difficulty ourselves and then involving a researcher from our lab who had never used AntTracks before as a pilot user. Based on the feedback, we adjusted the study method, e.g., we removed ambiguities in the instructions.

6.1 Study Subjects

We selected the study participants from two sources: (i) we invited students from a course on Java performance monitoring and benchmarking to take part in the study, as the course ensured basic knowledge about the context and purpose of memory analysis tools (cf. Ko et al.'s inclusion condition [59]). Ten students agreed to take part in the study. We made clear in the invitation that the participation and performance in the study are in no way related to the grading of the course. (ii) In addition, four researchers from our department accepted to participate. Table 3 shows a complete list of all participants. None of them had used AntTracks before, yet six of the 14 participant heard about AntTracks in presentations. At the time of the study, nine participants were in their bachelor's studies, two pursued a master, and three were enrolled in a PhD program. Their average experience in software engineering was 5.6 years whereas their average experience with memory analysis tools was 0.6 years. The ten students attending the Java performance monitoring and benchmarking course had completed one homework assignment involving the use of a memory analysis tool such as VisualVM [108] or MAT [106] to inspect the memory behavior of easyTravel [28], a state-of-the-art demo application by Dynatrace [29] that mimics the server of a travel agency. Seven of the students reported this assignment as their only memory analysis experience, which we regarded as an experience of 0.1 years. Besides VisualVM [108] and MAT [106], the participants already had experience in various tools including Java Melody [51, 69], Android Studio [37, 42], Valgrind [26, 79], and Java Mission Control [86] (including Java Flight Recorder [82]).

6.2 Study System

We selected the web application JPetStore 6 [78] as our study system. JPetStore has been widely used in research projects [32, 53, 55, 56, 112]. It models a minimalistic web shop for pets and uses a clearly structured class hierarchy. Categories (e.g., fish) can contain multiple products (e.g., Koi), which in turn can contain multiple items (e.g., spotted Koi and spotless Koi). Categories, products, and items each have their own web page and can be viewed in a web browser. We chose

Table 3. Study subjects' experience in software engineering and memory analysis.

#	Current Study	Experience in software engineering (in years)	Experience in memory analysis (in years)	Experience in AntTracks
1	Bachelor	3	0.1	No
2	Bachelor	6	0.1	No
3	Bachelor	4	0.1	No
4	Bachelor	3	0.1	No
5	Master	7	0.0	Presentations
6	Bachelor	1	0.1	No
7	Bachelor	2.5	0.1	No
8	PhD	7	2.5	Presentations
9	Bachelor	7	3	No
10	Master	5	0.1	Presentation
11	Bachelor	8	0.5	No
12	PhD	10	0.1	Presentations
13	Bachelor	6	1	Presentations
14	PhD	9	0.5	Presentations
AVG		*5.6*	*0.6*	

JPetStore since its straightforward structure described in a UML [13] class diagram made it easily comprehensible for the study participants without being familiar with the source code. This helped to mitigate the risk of participants not finishing the study tasks (cf. [118]).

To prepare the system for the study, we modified the JPetStore source code to seed two memory anomalies:

(1) *Memory Leak Mode*: In this configuration, we purposely keep objects alive after their intended use. Since memory leaks caused by a single object (e.g., a single static list) can easily be inspected and resolved by a dominator analysis [106], we mimic a more realistic problem that is harder to resolve due to multi-object ownership [114]. To achieve this, every time a product web page is requested, the (normally temporary) `Product` object shown on the page is stored in and kept alive by two different static lists located in different classes.
(2) *Memory Churn Mode*: In the original version of JPetStore, displaying a single item results in a database query using the item's ID, causing only a single `Item` Java heap object to be created. In our modified version, all available items are loaded from the database (a `List<Item>` of length 10 000) and the needed item is then extracted from this list. This means that 9 999 `Item` heap objects are needlessly created on every request, a typical case of high memory churn.

We created AntTracks trace files before the user study for both the memory leak and the memory churn mode. In particular, we simulated heavy load of the application via a script sending a large number of requests to the server, requesting random category web pages, product web pages, and item web pages.

6.3 Study Process and Data Collection

We conducted the study in a separate session with each subject. At the beginning of each session, we asked the participants to 'think aloud' [47, 50, 81] during the study. Specifically, we asked them to verbally describe what they were doing, to comment on any of their concerns, and to say whatever comes to their mind while solving the given tasks. A scribe documented the think-aloud statements, while a moderator watched and guided the subjects through the study and took additional notes on interesting observations not covered by the think-aloud protocol. Specifically, we conducted the following process that took approximately one hour per subject:

Preparation. Since the participants worked on the computer of one of the authors, all services and applications that might have distracted a study participant were closed. Before each session, we started the AntTracks Analyzer tool and loaded the trace file that was recorded using JPetStore in memory leak mode. The scribe and moderator prepared their documents to take notes. The moderator additionally prepared a utility questionnaire for the interview at the end of the session.

Briefing. The moderator explained the goals of the study to the participants and asked for their consent on the publishing of the findings [59]. Consent was given by signing a form explaining the study process, the data planned to be collected, and the procedures for storing and processing this data. The moderator also discussed a briefing sheet explaining the JPetStore domain, basic workflows of analyzing a memory leak and high memory churn, and the think-aloud process with the participants. The participants then received a document describing the different tasks to be performed. To ensure focus, the tasks were introduced one after another as the participants progressed through them.

Task Execution. The participants completed the following six tasks:

- *Memory Leak Detection:* They used AntTracks to detect suspicious memory behavior by inspecting the application's memory consumption over time in the Overview tab.

- *Heap Evolution over Time:* The participants selected a suspicious time window and used the TrendViz feature to identify the domain objects showing the highest memory growth in this interval. They further checked at which allocation sites these objects were created, and from which sites they were called.
- *Table-based Heap State Analysis:* The participants opened the heap state at the end of the selected time frame and performed a bottom-up analysis on the domain objects. They identified the objects showing suspicious memory growth in the previous analysis.
- *Graph-based Heap State Analysis:* As a next step, they used the graph visualization to explore the neighbors of these objects. Objects remain alive if they are (indirectly) referenced by GC roots. Participants were asked to follow the from-pointers, in particular, thick edges indicating a large ownership, to find suspicious GC roots. After this analysis, participants used the application's source code and aimed to fix the memory bug based on their findings.
- *Memory Churn Detection:* After fixing the memory leak, the participants analyzed the second trace file recorded in memory churn mode. They identified suspicious behavior by inspecting the memory charts in the application's details view.
- *Heap Evolution of Short-living Objects:* Participants were asked to select a time window showing suspicious GC behavior, i.e., frequent collections with high object death rates, for the short-living objects analysis. They were then asked to locate and fix the memory problem in the source code based on their findings.

Data Collection. After finishing all tasks the participants completed a usability questionnaire covering Nielsen's usability attributes [80] and specific capabilities of AntTracks. The moderator further conducted a semi-structured interview comprising 17 questions on the tool's utility and usability. The questions are based on the utility questionnaire by Davis [23] and usability issues revealed in the cognitive dimensions assessment (cf. Section 5). More specifically, we included one question per cognitive dimension classified as *Interview* in Table 2, such as 'Did you experience any problems while selecting a given time or time window?' (targeting *error-proneness* on the Overview and Details view) or 'Did you experience any problems with the used terminology, i.e., naming of displayed content and/or icons used?' (targeting cross-cutting *abstraction*). During these interviews, we also collected the demographic information presented in Table 3. Each interview was concluded with a short debriefing.

Data Analysis and Reporting. After running all sessions we prepared the collected data for analysis. Overall, 370 observations (26.5 per subject on average), 261 think-aloud statements (18.5 per subject on average) and 238 interview statements (17 per subject) were recorded by the scribes, some of which will be quoted in Section 7.2. We labelled all observations, statements and interview answers to allow their systematic use. For example, as shown in Table 4, the think-aloud statement on AntTracks' overview screen '*In the chart, I can see that my memory grows more and more, that is not good.*' received the labels '*Detects Growth In Chart*' as well as '*Recognizes Growth as Problem*'. We adopted and adjusted an iterative labelling process [35] that is similar to Open Coding [100]. First, a set of possible *observation labels* and *statement labels* had to be formed. For this, three of the authors jointly classified a sample set of the study session transcripts. This helped them to gain a mutual agreement on the possible labels and the coding process itself. Then each of the three authors individually coded the remaining observations and statements, while still staying in contact with each other. This allowed the coders to quickly and collectively decide if a new label should be introduced in case an observation or statement could not be mapped to an existing label. In this case, they also went through all previously labelled observations and statements to check if the new label should also be applied. Finally, the coders had a joint discussion meeting to merge the

Table 4. Labeling a think-aloud statement and linking these labels to their relevant cognitive dimensions.

Subj.	Task	Observation / Think-aloud statement	Labels	Relevant cognitive dimensions
1	1	'In the chart, I can see that my memory grows more and more, that is not good.'	Detects Growth In Chart Recognizes Growth as Problem	Visibility Hard mental operations

three individually labelled lists of observations and statements into a single list, thereby discussing and resolving possible differences.

Each of the labels was then linked to its relevant cognitive dimensions. For example, in Table 4 the two labels are related to the cognitive dimensions *visibility* and *hard mental operations* respectively. Discussing the result of the study (cf. Section 7) can then be done view by view, analyzing each cognitive dimension in question based on the frequency of relevant labels.

Table 5. Results of the usability questionnaires. We used a four-point scale (0, 1, 2, 3) for *learnability* (very hard to very easy), *error prevention* (too many errors encountered to no errors encountered), *subjective satisfaction* (very bad to very good), and *efficiency* (very inefficient to very efficient). For *memorability*, we used a yes/no question. Aggregations have been performed using the median.

Nielsen's Attr. / Subj.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	*Med.*
Learnability	**3**	**2.5**	**2**	**3**	**2.5**	**2**	**2.5**	**3**	**2**	**2.5**	**3**	**2**	**1.5**	**2.5**	***2.5***
Overview	3	3	3	3	2	1	2	3	2	2	3	2	2	3	*2.5*
TrendViz	3	2	2	2	2	2	3	3	2	2	3	2	1	3	*2*
Heap State	3	3	1	3	3	3	2	3	3	3	3	2	2	1	*3*
Graph	2	1	0	3	3	0	3	2	2	3	2	1	2	3	*2*
Details	3	2	3	3	2	2	3	3	3	3	3	3	1	2	*3*
Short-living Objects	3	3	2	2	3	3	2	2	2	2	2	2	1	2	*2*
Error Prevention	**3**	**2.5**	**3**	**3**	**3**	**3**	**3**	**2.5**	**3**	**3**	**3**	**3**	**3**	**3**	***3***
Overview	2	2	3	3	2	1	2	2	3	3	3	3	3	3	*3*
TrendViz	3	3	3	3	3	3	3	3	3	3	3	3	3	3	*3*
Heap State	3	3	3	3	3	3	3	3	3	3	3	3	3	3	*3*
Graph	3	2	3	3	3	3	3	3	3	1	3	3	3	3	*3*
Details View	3	2	3	2	2	2	3	2	3	3	3	3	3	3	*3*
Short-living Objects	3	3	3	3	3	3	3	2	2	3	3	3	3	3	*3*
Subjective Satisfaction	**3**	**2.5**	**2**	**3**	**2**	**2.5**	**3**	**3**	**3**	**2.5**	**3**	**2**	**2.5**	**2**	***2.5***
Overview	3	3	2	3	2	2	3	3	3	2	3	3	3	3	*3*
TrendViz	3	2	1	3	2	2	3	3	3	3	3	2	1	2	*2.5*
Heap State	3	3	2	3	3	3	3	3	3	3	3	2	3	1	*3*
Graph	3	1	2	3	3	1	3	3	3	3	3	2	2	2	*3*
Details View	3	2	2	3	2	3	3	3	3	2	3	3	3	3	*3*
Short-living Objects	2	3	2	3	2	3	3	2	3	2	2	2	2	2	*2*
Efficiency	**3**	**3**	**2**	**3**	**3**	**2**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	***3***
Memorability	**3**	**0**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	**3**	***3***

7 STUDY RESULTS

We discuss the results regarding the usability of the AntTracks Analyzer memory analysis tool based on the usability questionnaire and the findings for specific memory analysis tasks. We further report findings regarding utility. Recommendations derived from these results are then presented in Section 8.

7.1 Usability Questionnaire

Table 5 depicts the results of the usability questionnaire, which follows Nielsen's attributes of usability [80]. In the following, we will summarize the result for each of these attributes.

Learnability. In general, 13 of 14 subjects found the tool easy or very easy to learn. We see potential for improvements especially with regard to the learnability of the graph visualization, which was twice rated as hard to learn and twice as very hard to learn.

Subject S13, who managed to solve all tasks, surprisingly rated three of the six views used during the study as hard to learn. During the interview, it became apparent that this rating was mainly due to the subject's high familiarity with the MAT tool. MAT offers different views and analysis techniques than AntTracks, which led to some confusion. S3 assessed two of the views as (very) hard to learn, but repeatedly regarded his background in memory analysis as weak.

Error Prevention. Most participants replied that they did not encounter any errors. Participants reporting errors on the overview and the details view struggled with zooming and selecting time windows on the charts. The errors reported for the short-living objects concerned a minor visualization bug.

Subjective Satisfaction. The participants were, generally speaking, very satisfied with the tool. The basic views (overview, heap state table, and details view) had the highest ratings, while the more advanced views (TrendViz view, graph visualization and short-living objects view) were rated slightly lower. These satisfaction scores coarsely match the learnability scores.

An issue that may explain some of these lower ratings was brought up repeatedly during the study and the interviews: AntTracks was mainly perceived as a tool aimed at experienced users. Some novice study participants missed specific guidance that helps them to exploit the tool's full potential. Some users also lacked the background knowledge needed to correctly interpret the visualized data and gain insights from the presented metrics.

Efficiency. We asked all participants if they could productively use the tool in their daily work. Twelve of the 14 participants answered this question with *very efficient* and two with *efficient*. This supports our belief that by further increasing AntTracks' learnability, even novice users could use it efficiently to resolve memory anomalies in their applications.

Memorability. Thirteen of 14 study participants think that they will remember how to work with AntTracks after not using it for some time. Only one participant answered this question negatively.

7.2 Usability Results for Specific Activities

We now discuss the usability of the AntTracks Analyzer in detail by referring to the think-aloud statements (THA), observations (OBS), and the answers to the interview questions (INT) obtained during the study. As discussed in Section 5, we focus on a number of cognitive dimensions per view during this analysis. For each selected cognitive dimension we provide a *study result* (listed in Table 2), a statement summarizing the results, as well as a more detailed study report.

7.2.1 Memory Growth Detection — Overview.
The cognitive dimensions we studied in more detail on this view are the *hard mental operations* possibly needed to detect and interpret suspicious memory growth and the *error-proneness* of interacting with the time-series charts.

Although all participants could detect the memory growth, only 70% of them managed to select a good time window for analysis. Thirteen out of 14 participant mentioned that the application contains suspicious memory growth (THA). Similarly, during the interview, all 14 participants stated that they had no problem to detect the memory growth (INT). Nevertheless, we observed that this did not mean that users could also select a good window for analysis. As shown in Figure 1a, the memory chart shows a tall memory consumption spike during initialization – a typical memory pattern that is not related to a memory leak. Still, four of the 14 participants selected a time window covering the spike instead of the memory growth for further analysis (OBS), while two other participants expressed (THA) that they probably should investigate the spike in addition to the continuous memory growth. After the moderator explained that a time window covering the memory growth should be selected, over half of the participants expressed concerns regarding the optimal size of the time window (THA). (*Hard mental operations: Medium*)

Unintuitive controls caused problems with chart interaction. We observed that AntTracks' chart interactions confused most of the participants. The text next to the chart explaining the different ways of interaction was often ignored (OBS). Overall, only three out of 14 participants had no problems when interacting with the chart (INT). Rather, typical statements were '*The controls are not clear and the explanation text is too long*' (THA - S6), '*How do I select a time window?*' (THA - S1, S5), and '*Intuitively, I would have dragged for time window selection*' (INT - S4). Many participants rated this fact as an error, which is also reflected in the usability questionnaire (cf. Table 5). (*Error-proneness: Bad*)

The position of tabs and buttons led to irritations. Twelve out of 14 participants struggled to find the button to start the heap evolution analysis (OBS). The AntTracks version used in the study displayed the list of all open tabs and the list of available operations on the left side of the screen. Five people suggested to place the list of tabs and operations on the right side of the screen (INT) (as shown in Figure 1a), since '*It is typical to look for buttons in the bottom-right corner*' (INT - S5). (*Other findings - Visibility*)

7.2.2 Memory Growth Inspection: Evolution over Time — TrendViz View.
We studied five cognitive dimensions on the TrendViz view as shown in Table 2.

AntTracks default classifier combinations allowed novices to select suitable classifiers for heap object grouping. AntTracks' classifier selection system has been reworked before the user study to provide default classifier combinations to choose from, hiding its complex selection dialog in an expert mode. Except for two participants, one of which wanted to try the expert mode out of curiosity (OBS - S5, S13), all other participants selected the correct classifiers without problems by using one of the pre-defined classifier combinations. (*Diffuseness: Good; Viscosity: Good*)

Participants succeeded with finding the drill-down feature. We added interactive tables next to the charts before the study, to allow the users to drill-down by clicking on table entries. This paid off, as all participants except one discovered the drill-down feature (OBS). During the interview, we asked the participants whether they were aware of the drill-down feature in the view. A typical response was '*No. But I intuitively clicked on the table and expected something to happen.*' (INT - S1, S7, S9, S10, S11). One participant stated that he '*would probably have clicked on the chart if it contained a hover effect combined with a changed mouse cursor and a tool tip*' (INT - S4). (*Visibility: Medium*)

Participants understood the drill-down feature and sub-chart dependencies but were confused by terminology. Eleven subjects had no problem understanding the drill-down feature (INT), two subjects said it took them some time (INT - S6, S13), and one participant had general problems to understand the TrendViz view (OBS - S3). When asked whether they understand what happens when they drill-down, answers included '*More details of the selected elements are shown*' (INT - S2), '*It is like following one branch in a tree*' (INT - S4), and '*Selecting a type to show its allocation sites was clear to me*' (INT - S1). Thus, we can conclude that in general the meaning of the drill-down was clear to the subjects. Yet, we observed that four subjects stopped the drill-down early due to bad abstraction. Instead of investigating all available four call sites (by performing four drill-downs) they stopped at the first call site, i.e., '`(hidden internal call sites)`' (OBS), as they expected no additional call sites (THA). (*Abstraction: Medium; Role-expressiveness: Good*)

7.2.3 Memory Growth Inspection: Single Point in Time — Heap State View.

The cognitive dimensions *viscosity*, *diffuseness*, and *role-expressiveness* on this view are relevant for the classifier selection, as well as for the view in general. The problem of broken *hidden dependencies* was no longer an issue after fixing a potential problem before the study (OBS).

Half of the participants were unaware of the direct switch from heap evolution analysis to heap state analysis. Even though the TrendViz view offers easy switching to the heap state analysis at the end of the selected time window, seven participants were unaware of that feature and instead used a complex work around, i.e., going back to the overview screen and manually selecting this point in time on the chart (OBS). (*Viscosity: Medium – Navigation between views*)

The participants effectively used the pre-defined classifier combinations for typical heap state analysis tasks and selected them based on either name or description. In the interview we asked the participants about the selection of the classifier combination in the heap state view. Seven said that they selected the correct classifier combination based on its *name* while six answered that they used the textual *description* for selection (INT). Three participants stated that they additionally checked the set of classifiers to make sure it matches the description (THA). Only a single participant selected a wrong combination by ignoring all information and by just using the pre-selected default combination (OBS - S10). (*Diffuseness: Good; Viscosity: Good – Classifier Selection*)

Some participants regarded the tree table view as too diffuse. While inspecting the heap state, three participants mentioned that '*too much information is shown on the tree table view*' (THA - S3, S5, S14), and that they wished for a master-detail view (INT) to reduce the view's *diffuseness*. (*Diffuseness: Medium – Heap state table*)

Hiding the complexity of the classifier system made selection easier but half of participants no longer understood its full possibilities. It was obvious for half of the participants in the interview how the classification system works and that the pre-defined combinations just differ in terms of which filters and classifiers are applied (INT). Four said that they were aware of some classification system, understood the results intuitively, but did not think about how it works in the background (INT). Three others said that they did not recognize a classifier system as such but just selected a classifier combination name based on the instructions (INT). (*Role-expressiveness: Medium*)

Participants struggled with terminology. We again observed problems regarding the cognitive dimension *abstraction*, as some people struggled with terminology used on the heap state view. For example, two participants did not understand the meaning of '*hidden internal call site*' (THA - S5, S10), while one could not make sense of the term '*retained size*' (THA - S6). (*Other findings – Abstraction*)

7.2.4 Memory Growth Inspection: Single Point in Time — Graph View.
The object graph visualization in AntTracks enables users to visually explore the paths from a group of objects to their GC roots. We received valuable feedback during the study regarding this view.

Meaning of graph view elements not immediately clear. Only five subjects said that all operations and elements on the view were clear (INT). Four subjects said that they need better edge labeling (INT) to understand how many objects are kept alive on which path (THA, INT) and three subjects suggested to explain the operations in more detail, e.g., by using a tooltip (INT). One subject suggested to reduce the amount of text used in GC root nodes and to highlight different GC root types in different colors (INT - S8). Two participants generally struggled with using this view (OBS - S3, S6). For example, one participant did not understand that nodes and edges represent groups of objects and object references respectively but instead thought the graph would visualize relationships between outer and inner classes (THA - S6). (*Role-expressiveness: Bad*)

Graph complexity regarded managable. Even though the graph view displays many elements, no participant expressed any concerns, possibly also due to AntTracks' highlighting of important edges with thick lines to guide the users. This is further confirmed by the fact that half of the participants liked this edge highlighting (THA, INT). (*Diffuseness: Good*)

Interaction in graph-based visualization worked fine. As this is the only view in AntTracks that does not use tables or time-series charts for visualization, we studied whether participants have problems with this break in consistency. However, all subjects managed to apply the operations and use the features for zooming and panning, and no problems with the interaction mechanisms were found (OBS, THA). (*Consistency: Good*)

Half of the participants struggled to extract the information needed to fix the memory leak. Finding the most suspicious GC roots relies on detecting those paths on which the most objects are kept alive. However, as already mentioned, edge labeling was a source of confusion (OBS, THA). Also, two subjects (correctly) expressed concerns whether thread-local variables (that were at that time visualized in the same way as the more dangerous static fields) should also be considered as suspects (THA). In addition, we observed that three participants ignored the fact that the memory leak was caused by multi-object ownership, i.e., two different static fields kept the same objects alive, even though they were both shown side by side on the graph view (OBS). (*Hard mental operations: Bad*)

Two thirds of the participants managed to trace the memory leak to source code. Overall, six of the 14 participants were able to identify both static lists that are responsible for the memory leak within the source code (OBS). Three participants found one of the two sources for the memory leak, but did not recognize the multi-object ownership in the graph view (OBS). Five participants were not able to use the insights gained in the tool to make the necessary fixes in the source code (OBS). (*Other findings*)

7.2.5 Memory Churn Detection — Details View.
The second part of the study involved the analysis of memory churn. The participants' first task was to inspect the development of the application's memory footprint over time and to look for memory anomalies using AntTracks' details view. Since this view uses the same charting technique as the overview, the problems regarding the *error-proneness* of chart interactions reported in Section 7.2.1 also affected this view.

Novices struggled with correctly interpreting the spike pattern. The main goal on the details view is to detect memory churn hotspots based on spike patterns on the memory chart, as shown on Figure 2a. Twelve of 14 participants recognized this spike pattern (THA). Two participants wrongly focused on the initial memory spike as discussed in Section 7.2.1 (OBS - S1, S3). Five

participants had no idea whether such spike patterns are abnormal and have to be inspected further (THA). The statements ranged from '*I think many short-living objects are normal in Java applications*' (THA - S12, S13) to '*It cannot be normal to allocate two million objects when only showing a few webpages*' (THA - S11). Selecting a good time window for the analysis was also hard for most participants as the size of the selected time windows varied drastically from 2 to 200 garbage collections (OBS). Very short or very long time windows may distort the analysis result, and a very large time window may unnecessarily increase the analysis time. (*Visibility: Good; Hard mental operations: Bad*)

GC-specific terminology confused some participants. The details view shows three different series per memory chart, representing the different generations of the GC: *Eden*, i.e., objects that are new and have not yet survived a single garbage collection, *Survivor*, i.e., objects that have survived a few garbage collections, and *Old*, i.e., objects that have lived for a long time and are stored separately in the heap in order to speed up frequent garbage collections. Seven participants stated that these terms are either unknown to them or that they once learned about them but have forgotten them (THA, INT). (*Other findings*)

7.2.6 Memory Churn Inspection: Evolution over Time — Short-living Objects View.

The upper half of the short-living objects view presents pie charts showing the types and allocation sites that caused the most garbage in terms of objects and bytes in the selected time window (cf. Figure 2b). The bottom half of the view shows charts comparing the selected time window's GC frequency and GC overhead with the application's average. During the study, we studied the *role-expressiveness* and *diffuseness* of these charts. Additionally, the view provides a tree table view for detailed analysis which we inspected regarding *consistency*.

All participants managed to extract the information needed to inspect the problem despite some irrelevant and overly complex charts. All subjects managed to use the charts to recognize the type that caused the most garbage (OBS). The allocation site pie charts were, in general, understood but ignored (OBS, THA). Only one participant said that the allocation site charts are unclear (INT - S13). Yet, four participants expressed that they did not understand the charts on the bottom half of the screen (INT), while six other participants ignored these charts because they considered them as not relevant to the problem (INT). (*Role-expressiveness: Medium; Diffuseness: Bad*)

The participants had no problems with interpreting the tree table. After inspecting the charts, the participants switched to the tree table to inspect the garbage created over the selected time window. Since tree tables in AntTracks are generally used to visualize *live objects*, we wanted to know whether the study participants understood that *dead objects* are shown in this case. Thirteen participants stated that the content of the tree table was clear to them (INT). Only one participant replied that he '*had to think a bit about what the columns mean*' (INT - S8). (*Consistency: Good*)

Most participants could locate the memory churn location in the source code, six could to fix the problem. While looking for the root cause of the memory churn, all subjects started to inspect the top-most table entry, i.e., `char` arrays that died without surviving a single garbage collection (OBS). Since domain types (e.g., types that are not part of the Java library) are often easier to locate in the source code, AntTracks highlights these types in bold to direct the users' attention to these tree table entries (for example, in Figure 1d the type `Product` is highlighted as a domain type). Still, only half of the participants noticed the domain object type that generated the most garbage (OBS). During the interview, one participant stated that he would '*probably have investigated the domain objects if they were not only written in bold, but shown in a separate section within the tree table, or even as a separate pie chart on the overview*' (INT - S3).

Overall, eleven participants found the problematic method in the source code: all seven participants who inspected the domain objects in AntTracks and four of the seven subjects who did not inspect the domain objects in AntTracks. Only six of the eleven subjects who managed to locate the problematic method were also able to fix the problem. Five of them had inspected the domain objects (OBS). Subjects who did not focus on the domain objects in AntTracks were looking for allocations of `char` arrays (THA), while the others were looking for `Item` objects (THA). Many `Item` objects are created by the problematic code, but since every `Item` object contains up to eight `String` objects each again containing a `char` array, all these allocations are represented by the same call site, which obviously confused some of the subjects (OBS). (*Other findings*)

7.2.7 CD Assessment: Cross Cutting.

Several cognitive dimensions are relevant on all views or relate to the general structure and organization of AntTracks.

Constraints on the order of doing things. Participants did not feel constrained by the tool, which allows to go back after making a wrong step and to revert earlier actions (OBS). (*Premature commitment: Good*)

Basic layout and stacked tabs. We did not detect any hidden dependencies the participants were not aware of (OBS). When asked what they liked about the tool in general, six participants said that they like the general layout using stacked tabs and the consistent positioning of tabs and actions on the screen (INT). (*Visibility: Good; Hidden dependencies: Good*)

Assumptions made regarding terminology knowledge. We noticed that especially novice users occasionally struggled with terminology (OBS). For instance, half of the participants did not know the difference between eden, survivor and old objects on the details view (INT). Two participants did not understand the difference between an allocation site and a call site (INT - S7, S12) – a problem also affecting other participants as we suspect based on our observations (OBS). (*Abstraction: Bad*)

Consistency of visualizations. The participants did not point out inconsistencies regarding different means of visualization in AntTracks (INT). AntTracks' chart syncing, i.e., keeping the same zoom levels and time selections across multiple charts and views, was positively mentioned with regard to consistency (INT - S10). (*Consistency: Good*)

Suggestions for reducing complexity. We interviewed the study participants whether they considered any of AntTracks' views or interface elements as unnecessary or overly complex. Eight participants raised no concerns in this regard (INT). The suggestions for reducing complexity encompass a master-detail view on the heap state view (INT - S3, S14), the simplification and improved guidance on the graph visualization (INT - S6, S8, S12), and the removal of some charts on the short-living objects view (INT). On the positive side, two users commended AntTracks' good default settings while keeping expert modes for advanced users (INT - S4, S9). (*Diffuseness: Bad*)

7.3 Utility

To assess the utility of AntTracks' views, we first asked subjects whether they liked the flow suggested by the study, i.e., to first search for memory growth, then visualize the growth using the AntTracks TrendViz, and then inspect the final heap state followed by an analysis of the heap object graph. Twelve subjects found the order natural (INT), while two participants stated that they did not really need the trend analysis (INT - S2, S8). However, we also received positive feedback for the trend analysis, for example '*In hindsight, I liked the trend view. Without the study tasks, I would have directly selected a single point in time because I am used to it*' (INT - S4).

We also asked the participants whether they missed a specific feature or whether they wanted to make any other comments, resulting in a catalog of 19 feature requests and possible improvements

for AntTracks. The most-wished feature mentioned by five study participants concerns '*better guidance*' (INT). This is related to three sub-problems: (1) deriving knowledge from the current screen, i.e., interpreting the shown data; (2) selecting appropriate steps which should be taken next based on the findings on the current view; and (3) guidance within the IDE on how to fix problems in the source code. Three participants wished for '*more tool tips*' across the various views, especially on the graph view (INT). Some improvements were suggested by at least two participants, such as a '*master-detail view*' for views using tree tables (INT - S3, S5), as well as tutorial videos explaining how to use the tool (INT - S3, S6). These videos could be uploaded to a video platform (such as Youtube), as well as directly integrated into AntTracks for ad-hoc learning / learning by doing [99] (INT). All other suggested improvements were suggested by a single subject and describe rather minor changes, often with potentially high impact, e.g., zooming with CTRL-key and mouse wheel (INT - S5) or highlighting the currently hovered line in the tree views for better orientation (INT - S8). There was also a wish for more automated analyses (INT).

Finally, we asked the study participants what they liked and disliked about the tool in general. Most participants mentioned that they liked AntTracks' general look-and-feel as well as its layout and visual structure (INT). Four participants said that they liked the charts, i.e., their structure, labeling and the selection synchronization feature (INT). Two participants mentioned AntTracks' high productiveness (INT) and also found it intuitive to use (INT). The most disliked aspect of the tool was the chart interaction. Five participants mentioned that they did not like the way how zooming and time window selection works (INT), two participants pointed out that AntTracks and especially its charts react laggy to user input (INT). Four participants said that, although they liked the way how the tabs and operations are arranged, they should be placed on the right side of the window since this is where most users look for operation buttons (INT). As already mentioned it earlier, three participants repeated that they lacked guidance for novice users (INT).

8 RECOMMENDATIONS

Based on the study results presented in the previous section we derived nine general recommendations for developers of memory analysis tools. Table 6 shows each of these recommendations as well as cross references to the study results based on which we formulated the recommendation. In the following, we shortly describe each of them.

Use flexible drill-down mechanisms. AntTracks heavily relies on its classification system with multi-level grouping [119, 122]. During the study, we were able to observe that though not all participants recognized the classification system as such even novice users were able to easily understand the resulting tree-structured data. Allowing the users to drill down on this data level by level (e.g., from the whole heap to the most allocation intensive type further to this type's most allocation intensive allocation site) enables them to focus on one thing at a time. The same could be observed on the graph view. The participants performed best when they could focus on one task, inspecting one path to a GC root at a time, drilling down step by step.

Hide complexity using task-specific default settings. Even though this may sound trivial, we can clearly see that the study participants profited from pre-defined classifier combinations for different typical analysis task. Before introducing these pre-defined combinations, AntTracks also pre-selected a default combination, yet this combination had to be slightly adjusted based on the task's analysis goal - a seemingly simple task which nevertheless can demand too much from novice users. Thus, we suggest to define typical tasks and workflows and provide default settings for each of them.

Carefully select and explain memory analysis terminology. Probably one of the most clear results of the study is that AntTracks used too much terminology that is unknown to novice users (such as *allocation site*, *call site*, *retained size* or the garbage collector's space names *eden*, *survivor*, and *old*). Tool developers should aim to improve this situation by reducing the number of complex terms in situations where they are not really needed and by providing easy-to-understand explanations.

Show details and advanced analysis results only on demand. As stated above, the participants performed best when they were able to focus on a single task, drilling down on the problem step by step. This also relates to the well-known Shneiderman's mantra '*Overview First, Zoom and Filter, Details on Demand*' [101]. By first giving an overview and only showing details on demand, the view's diffuseness is reduced and its visibility is increased. This does not only concern the analysis views, but also complex configuration views or settings panes. For example, AntTracks by default hides the expert mode on the classifier selection view, yet offers very flexible configuration possibilities on demand. Other examples in AntTracks encompasses the *Overview* screen, which shows only two charts, both of which only contain a single chart series, to be as easy to understand as possible. For more detailed information, the user can switch to the *details view*. A counter-example is AntTracks' *short-living objects view*, which contained far too many charts. Since many of these charts are not needed to find the root cause of high memory churn but rather present

Table 6. Summary of Recommendations

Name of recommendations	**Cross references to relevant study results**
Use flexible drill-down mechanisms	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Graph view (Section 7.2.4), Short-living objects view (Section 7.2.6)
Hide complexity using task-specific default settings	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Cross cutting (Section 7.2.7)
Carefully select and explain memory analysis terminology	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Details view (Section 7.2.5), Cross cutting (Section 7.2.7)
Show details and advanced analysis results only on demand	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3), Short-living objects view (Section 7.2.6), Cross cutting (Section 7.2.7)
Provide support for time selection in large time series	Overview (Section 7.2.1), Details view (Section 7.2.5)
Ensure a smooth transition from evolution analysis to snapshot analysis	TrendViz view (Section 7.2.2), Heap state view (Section 7.2.3)
Use automation to relieve users from complex tasks	Overview (Section 7.2.1), Details view (Section 7.2.5), Graph view (Section 7.2.4)
Provide guidance and explanations to support exploratory learning of analysis capabilities	Overview (Section 7.2.1), Graph view (Section 7.2.4), Details view (Section 7.2.5), Cross cutting (Section 7.2.7)
Provide IDE integration to guide diagnosis of memory bugs	Graph view (Section 7.2.4), Short-living objects view (Section 7.2.6)

general and additional information on the GC activity, they should be moved to another tab to reduce diffuseness.

Provide support for time selection in large time series. This recommendation is two-fold: First, it suggests to use well-known and established user input handling on time-series charts, and second it suggest to provide tool support for intelligent (semi-)automatic selections. For example, regarding the first recommendation, study participants suggested to use input handling similar to Audacity[3]. This application provides a wide array of interaction possibilities such as the selection of time windows by dragging the mouse, where these time windows can be further modified, for example moved or resized. Regarding the second recommendation, by analyzing the chart's underlying time series data [34], the tool can automatically detect suspicious patterns (such as continuous memory growth or frequent memory spikes). When such a pattern is detected, the tool can then suggest a time windows for memory analysis inspection to the user [116], further reducing the need for hard mental operations.

Ensure a smooth transition from evolution analysis to snapshot analysis. In general, we can distinguish two types of analyses that can be performed in memory analysis tools: Analyses that inspect the application at a given point in time, and analyses that inspect the application's behavior over time in a given time window. Both of these analysis types are often interwoven and follow each other in a typical analysis workflow. For example, when investigating memory leaks, the user typically first inspects which objects accumulate *over time* in a certain window, and then inspects the object graph around these objects at the end of the time window, i.e., at a given *point in time*, to find the reason for their accumulation. By defining typical workflows, i.e., typical orders of analysis steps, users can be better guided through the whole process, making transitions between different types of analysis easier to follow.

Use automation to relieve users from complex tasks. This recommendation mainly focuses on reducing the amount of hard mental operations. During the study, especially on the graph view, the participants expressed desire for automation to reduce the amount of manual work needed during the analysis. For example, they suggested that paths to the GC roots could automatically be calculated and analyzed, only showing those most likely involved in a potential memory leak. The previously mentioned automatic suggestion of suspicious time windows also relates to this recommendation.

Provide guidance and explanations to support exploratory learning of analysis capabilities. Another main finding of the study is that novice users, even though often able to 'see' suspicious patterns, cannot correctly interpret them and thus cannot derive the right conclusions. This problem already became apparent during the first task of the study, in which the users had to inspect a memory time-series chart and select the most suspicious time window. While all of the participant saw the continuous memory growth there, 30% of the participants chose another time window. Also, one of the first questions by a participant was '*Is there some sort of suggestion available?*'. Similar interpretation problems could be observed throughout the study. We thus suggest that tools should not only provide the functionality to inspect memory anomalies, but they also need to provide guidance [33] to support exploratory learning and learning-by-doing [99] and to increase the tool's general learnability [1, 80]. Such guidances exist in various forms, ranging from simple wizards [27, 109] to context-sensitive help systems such as coaches, guides or advisors [27, 72]. For example, advisors are context-sensitive help systems that usually include hints, tips, reasoning support, and explanations of complicated concepts, helping novice users to make decisions and

[3] Audacity [3, 68] is a free, open source, cross-platform audio software.

helping to understand why a certain step must be performed or to determine why a certain decision was suggested. Such help systems, in combination with more recent approaches such as micro-learning and gamification [38, 43, 49], are often used during the process of onboarding, i.e., introducing a new tool or service to a person to ensure success [94, 103].

Provide IDE integration to guide diagnosis of memory bugs. Finally, we suggest a stronger integration of memory analysis tools with IDEs. Currently, most memory analysis tools are standalone applications, decoupled from the user's development environment. Yet, it is this development environment where the user is expected to fix the just detected memory anomaly, however, without further guidance. Developing an IDE plugin [4, 21] (and probably even expanding the tool's capabilities by the use of hybrid static and dynamic analysis [31]) would make it possible to automatically detect and highlight suspicious code segments and provide further guidance on the source code level.

9 PLANNED IMPROVEMENTS FOR ANTTRACKS BASED ON RECOMMENDATIONS

The previous section distilled a set of recommendations based on the results of our study. We now report our improvement plans for five areas of AntTracks based on these recommendations.

9.1 Allocation Site Handling

In complex software systems, call chains can become confusingly long. To mitigate this problem, AntTracks already distinguishes *domain call sites* likely part of the application under investigation and *non-domain call sites* in libraries, for example classes located in packages `java.*` or `javax.*`. Call chains stretching over multiple non-domain call sites are collapsed into a single entry named `hidden internal call sites`. In JPetStore, the test system inspected in the study, this often shortens call chains containing more than 25 entries down to five entries. Unfortunately, we observed that some participants still struggled with interpreting this entry and they also had problems to distinguish the terms allocation site and call sites. This means that the presentation of allocation sites and their call chains has to be further improved as expressed by our recommendation to *carefully select and explain memory analysis terminology*. Non-domain call sites could be hidden completely by default, only displaying them on demand, thereby implementing our recommendation to *show details and advanced analysis results only on demand*. For domain call sites, the call distance could be shown next to them, i.e., either *directly* called by X or *indirectly* called by X. Another planned feature is to show the call chain not textually but visually using a graph. Such a call graph could also be combined with techniques for hiding internal call sites (discussed above). Finally, a stronger coupling between the analysis tool and the IDE would realize our recommendation to *provide IDE integration to guide diagnosis of memory bugs*. Selecting an allocation site in the analysis tool could highlight the source code location in the IDE. We expect that this will help users to interpret the call hierarchy more easily as compared to just looking at the allocation site information in the analysis tool.

9.2 Domain Filtering

The idea of distinguishing non-domain call sites and domain call sites can also be applied to types, thereby distinguishing non-domain types and domain types. Our study showed that participants focusing on domain call sites and domain types were more successful in resolving the underlying problem in the source code. In AntTracks, domain sites and domain types are currently written in bold to gain the users' attention. Another technique to highlight domain objects is to automatically group non-domain objects and domain objects, thereby implementing our recommendation to *use automation to relieve users from complex tasks*. However, not every memory anomaly may be

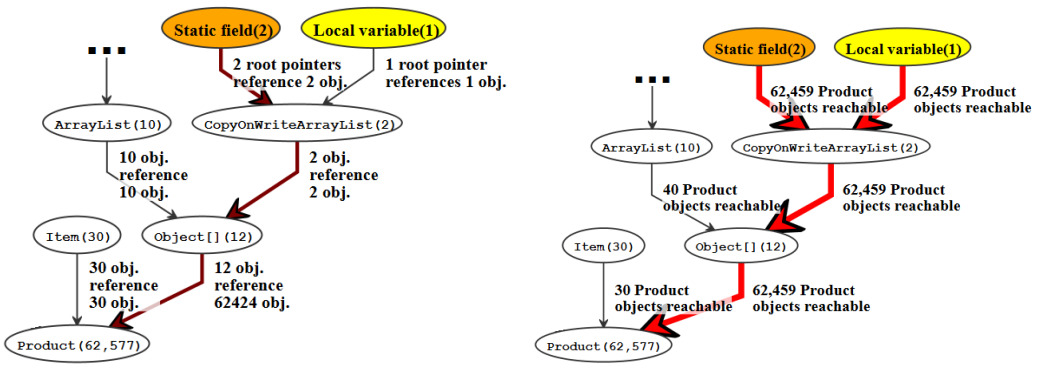
resolved by just looking at domain objects. For example, applications that mostly work with built-in data types may not profit from such a feature. Nevertheless, following our recommendation to *hide complexity using task-specific default settings* we think that inspecting the domain objects first reduces task complexity, in many cases already provides useful results, and sometimes even reveals the root cause of the underlying problem.

9.3 Chart Interaction Behavior

Many study participants were dissatisfied with the way AntTracks handles the zooming and the selection of time windows on time-series charts. The decision to use this convention was partially based on the default zooming behavior of the JFreeChart charting library [36], which defines zooming as mouse drag. We will thus improve AntTracks' chart interactions, thereby implementing our recommendation to *provide support for time selection in large time series*. To this end, we will study other tools and applications with similar chart interaction techniques (such as Audacity [3]) and take them as example.

9.4 Graph-based Views

Most state-of-the-art tools rely heavily on the visualization of data using (tree) tables. Yet, ample scientific work exists on more advanced features for memory visualization [2, 46, 71, 74, 93, 98, 129]. AntTracks thus already provides a graph-based visualization of the aggregated object graph to inspect the paths to the GC roots. The study revealed interesting findings in this regard: Firstly, GC roots were visualized in the same way regardless of their kind. However, thread-local variables are often very short-living and thus in most cases not relevant for identifying memory leaks. Visualizing such roots in the same way as more suspicious roots such as long-living static fields can distract tool users. We thus plan to use different notations to better highlight more suspicious GC roots. Secondly, the edge labels currently show how many objects from a given edge's start node reference objects of the given edge's target node. This makes it easy to inspect the ownership relationship between two neighboring nodes, but it remains hard to extract the ownership influence between two more distant nodes. For example, in Figure 4a, we see a part of the graph visualization the participants were confronted with during the study. It shows that nearly all `Product` objects (bottom-most node) are referenced by twelve different `Object[]` objects, two of which are referenced by `CopyOnWriteArrayList` objects and ten are referenced by `ArrayList` objects. Taking only edge



(a) Neighborhood labeling.

(b) Ownership labeling.

Fig. 4. Comparison of two edge labeling techniques.

labeling into account, this notation is not helpful to understand if the `CopyOnWriteArrayList` or the `ArrayList` objects keep alive more `Product` objects. To this end, we also encoded the ownership, i.e., how many objects are kept alive by a certain node, into the edge color and width. Even though the thicker colored edges guided the participant into the right direction, some reported that they did not understand why certain edges were colored and wider than others. This suggests that edge labeling and edge highlighting in memory object graphs should not be too general but problem-driven, as suggested by our recommendation *hide complexity using task-specific default settings*. For example, we meanwhile already realized a new graph view in AntTracks supporting the visualization of ownership edges, see Figure 4b. Instead of labeling edges based on the relation between two neighboring nodes, it takes one node, i.e., object group, and adjusts all edges and their labels to highlight those that keep alive most of the selected objects. The labels in Figure 4b now clearly highlight the path on which most `Product` objects are owned, i.e., kept alive.

9.5 Guidance

Many study participants expressed the risk of not using the tool to its full potential due to their missing background in memory analysis. They suggested that the tool should provide guidance features such that even novice users could use it without prior training. Since this was the most requested feature by the study participants, we have meanwhile extended AntTracks with an advisor feature realizing our recommendation to *provide guidance and explanations to support exploratory learning of analysis capabilities*. This guidance system performs four support operations on each analysis step: it (1) *detects* and (2) *highlights* the most important information on the screen, (3) *explains* why this information is important, and (4) *suggests* which next steps are appropriate based on these findings. This way, even without prior tool experience, AntTracks now guides users through the whole analysis process. For example, on the overview and the details views, AntTracks now automatically detects suspicious time windows, highlights them, explains why these time windows are considered suspicious and which next analysis steps seem appropriate [116], additionally realizing our recommendation to *provide support for time selection in large time series*.

Yet, while AntTracks now may guide users to suspicious allocation sites, they still have to fix the memory bug in their IDE without any further guidance. We thus further plan to integrate parts of this guidance also into IDEs, closing the gap of guided analyses in AntTracks and unguided source code inspection in the IDE, following the recommendation to *provide IDE integration to guide diagnosis of memory bugs*.

10 THREATS TO VALIDITY

A threat to construct validity is that our results might be biased due to the system we used during the study. However, we selected JPetStore because it easy to understand and at the same time allows to comprehensively assess the key views of AntTracks. Furthermore, JPetStore has also been used in many other studies [32, 53, 55, 56, 112] since it represents a clearly organized and realistic web application. Another threat to construct validity is the selection of appropriate tasks for our user study. To select representative memory analysis tasks, we discussed typical memory analysis activities by studying related work and state-of-the-art tools and selected the study tasks based on these activities. We further conducted a pilot study with a PhD student from our lab to further improve the expressiveness of the tasks.

To ensure external validity, we selected an interactive tool that is representative for the domain of memory monitoring and analysis. Although the implications we derived from the study depend on our experiences in using and working with AntTracks, the activities and capabilities are common in other memory analysis tools, as discussed in Section 3. Another threat is the use of students as subjects. They were selected based on their participation in a university course that teaches the

basics of monitoring and performance analysis and includes a homework assignment on memory analysis. This allowed us to ensure that all subjects have a minimum of background knowledge on the context of the study. We have also shown that most of the subjects are software developers with several years of experience. Furthermore, it has been argued that the differences between students and professionals are only minor if they perform relatively small tasks of judgment [48]. While the participation of additional experienced memory analysts may have added further value, we realized that the findings for our subjects nicely converge and that there were many common insights. When qualitatively analyzing our results we noticed a certain degree of saturation and the findings showed that adding more novice participants would not have led to more insights.

With regard to internal validity, the analysis of the collected data still depends on our own interpretation. However, this work was performed by three researchers and their results were checked by a fourth senior researcher. We also had regular joint meetings to reconcile different interpretations.

Regarding conclusion validity, the threat consists in the fact that our results are based on qualitative data [100]. Given that an aim of the study was to investigate the behavior and opinions of tool users, qualitative research methods are well-suited. We further applied the derived recommendations to AntTracks to additionally check their validity.

11 CONCLUSIONS

This paper presented the detailed results of assessing the usefulness of memory analysis capabilities as implemented in the AntTracks Analyzer tool using the cognitive dimensions (CD) of notations framework and a user study involving 14 subjects. The CD framework serves as a discussion tool for designers and people who evaluate designs of interactive artifacts, including software tools. We used the results of the CD assessment and the user study to discuss both specific implications for AntTracks as well as general recommendations for memory analysis tool developers. Practitioners and researchers can use our work as one example of how to assess the usability and utility of interactive memory analysis tools.

Overall, we can conclude that the usability and utility of AntTracks were perceived as very good. All subjects liked the tool and most of them were able to complete the tasks within the planned time. However, our observations revealed some issues with regard to diagnosing and fixing problems in the source code. Overall, the positive feedback is encouraging given the participants' limited background in memory analysis, their missing familiarity with the tool, and the complexity of the given analysis tasks. The comments and suggestions made by the study participants have been of great help to us to further improve our tool. For example, the guidance and support of novice users requested by many participants is a major focus for our ongoing work. First steps in this direction have already been taken since the study by introducing a system that automatically detects suspicious time windows, highlights them and explains why these time windows are considered suspicious [116]. This frees the users from the detection task, teaches them why a certain time window is of interest, and thus leads to a learning-by-doing effect [99].

In the future we will further improve our tool, especially regarding support for novice users, and conduct experiments with other systems and users.

ACKNOWLEDGMENTS

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

REFERENCES

[1] Alain Abran, Adel Khelifi, Witold Suryn, and Ahmed Seffah. 2003. Usability Meanings and Interpretations in ISO Standards. *Software Quality Journal* 11, 4 (2003), 325–338. https://doi.org/10.1023/A:1025869312943

[2] Edward Aftandilian, Sean Kelley, Connor Gramazio, Nathan P. Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Heapviz: interactive heap visualization for program understanding and debugging. In *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010*. 53–62. https://doi.org/10.1145/1879211.1879222

[3] Audacity. 2020. *Audacity: Free, open source, cross-platform audio software.* https://www.audacityteam.org/

[4] Sebastian Baltes, Peter Schmitz, and Stephan Diehl. 2014. Linking sketches and diagrams to source code artifacts. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-22)*. 743–746. https://doi.org/10.1145/2635868.2661672

[5] Matt Bellingham, Simon Holland, and Paul Mulholland. 2014. A cognitive dimensions analysis of interaction design for algorithmic composition software. In *Proceedings of the 25th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2014, Brighton, UK, June 25-27, 2014*. 18. http://ppig.org/library/paper/cognitive-dimensions-analysis-interaction-design-algorithmic-composition-software

[6] Verena Bitto and Philipp Lengauer. 2016. Building Custom, Efficient, and Accurate Memory Monitoring Tools for Java Applications. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. 321–324. https://doi.org/10.1145/2851553.2858664

[7] Verena Bitto, Philipp Lengauer, and Hanspeter Mössenböck. 2015. Efficient Rebuilding of Large Java Heaps from Event Traces. In *Proceedings of the Principles and Practices of Programming on The Java Platform, PPPJ 2015, Melbourne, FL, USA, September 8-11, 2015*. 76–89. https://doi.org/10.1145/2807426.2807433

[8] Alan Blackwell and Thomas Green. 2003. CHAPTER 5 - Notational Systems—The Cognitive Dimensions of Notations Framework. In *HCI Models, Theories, and Frameworks*. Morgan Kaufmann, San Francisco, 103 – 133. https://doi.org/10.1016/B978-155860808-5/50005-8

[9] Alan F. Blackwell. 2005. Cognitive Dimensions of Notations. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), 21-24 September 2005, Dallas, TX, USA*. 3. https://doi.org/10.1109/VLHCC.2005.26

[10] Alan F. Blackwell. 2008. Cognitive Dimensions of Notations: Understanding the Ergonomics of Diagram Use. In *Diagrammatic Representation and Inference, 5th International Conference, Diagrams 2008, Herrsching, Germany, September 19-21, 2008. Proceedings*. 5–8. https://doi.org/10.1007/978-3-540-87730-1_4

[11] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong, and R. Michael Young. 2001. Cognitive Dimensions of Notations: Design Tools for Cognitive Technology. In *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings*. 325–341. https://doi.org/10.1007/3-540-44617-6_31

[12] Michael D. Bond and Kathryn S. McKinley. 2006. Bell: bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. 61–72. https://doi.org/10.1145/1168857.1168866

[13] Grady Booch, James E. Rumbaugh, and Ivar Jacobson. 1999. The Unified Modeling Language User Guide. *J. Database Manag.* 10, 4 (1999), 51–52.

[14] Eric Bruneton, Eugene Kuleshov, Andrei Loskutov, and Rémi Forax. 2020. *ASM.* https://asm.ow2.io/

[15] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems* 30, 19 (2002).

[16] Kung Chen and Ju-Bing Chen. 2007. Aspect-Based Instrumentation for Locating Memory Leaks in Java Programs. In *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China, July 24-27, 2007. Volume 2*. 23–28. https://doi.org/10.1109/COMPSAC.2007.79

[17] Shigeru Chiba. 2020. *Javassist.* https://www.javassist.org/

[18] Shigeru Chiba and Muga Nishizawa. 2003. An Easy-to-Use Toolkit for Efficient Java Bytecode Translators. In *Generative Programming and Component Engineering, Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003, Proceedings*. 364–376. https://doi.org/10.1007/978-3-540-39815-8_22

[19] Adriana E. Chis. 2008. Automatic detection of memory anti-patterns. In *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*. 925–926. https://doi.org/10.1145/1449814.1449911

[20] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of Memory Inefficiency. In *ECOOP 2011 - Object-Oriented Programming - 25th European Conference, Lancaster, UK, July 25-29, 2011 Proceedings*. 383–407. https://doi.org/10.1007/978-3-642-22655-7_18

[21] Jürgen Cito, Philipp Leitner, Christian Bosshard, Markus Knecht, Genc Mazlami, and Harald C. Gall. 2018. PerformanceHat: augmenting source code with runtime performance traces in the IDE. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE 2018)*. 41–44. https://doi.org/10.1145/3183440.3183481

[22] Karl Cox. 2000. Cognitive Dimensions of Use Cases: Feedback from a student questionnaire. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*. 8. http://ppig.org/library/paper/cognitive-dimensions-use-cases-feedback-student-questionnaire
[23] Fred D. Davis. 1989. Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. *MIS Quarterly* 13, 3 (1989), 319–340. http://misq.org/perceived-usefulness-perceived-ease-of-use-and-user-acceptance-of-information-technology.html
[24] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. 2002. Visualizing the Execution of Java Programs. In *Software Visualization*. 151–162. https://doi.org/10.1007/3-540-45875-1_12
[25] Wim De Pauw and Gary Sevitsky. 2000. Visualizing reference patterns for solving memory leaks in Java. *Concurrency - Practice and Experience* 12, 14 (2000), 1431–1454. https://doi.org/10.1002/1096-9128(20001210)12:14<1431::AID-CPE542>3.0.CO;2-2
[26] The Valgrind Developers. 2020. *Valgrind*. http://valgrind.org/
[27] D. Christopher Dryer. 1997. Wizards, Guides, and beyond: Rational and Empirical Methods for Selecting Optimal Intelligent User Interface Agents. In *Proceedings of the 2nd International Conference on Intelligent User Interfaces, IUI 1997, Orlando, Florida, USA, January 6-9, 1997*. 265–268. https://doi.org/10.1145/238218.238347
[28] Dynatrace. 2017. *Demo Applications: easyTravel*. https://community.dynatrace.com/community/display/DL/Demo+Applications+-+easyTravel
[29] Dynatrace. 2020. *Dynatrace*. https://www.dynatrace.com/
[30] ej technologies. 2020. *JProfiler*. https://www.ej-technologies.com/products/jprofiler/overview.html
[31] Michael D. Ernst. 2003. Static and Dynamic Analysis: Synergy and Duality. In *Workshop on Dynamic Analysis (WODA '03)*. Portland, OR, USA, 24–27.
[32] Florian Fittkau, Phil Stelzer, and Wilhelm Hasselbring. 2014. Live Visualization of Large Software Landscapes for Ensuring Architecture Conformance. In *Proceedings of the ECSA 2014 Workshops & Tool Demos Track, European Conference on Software Architecture, 2014, Vienna, Austria*. 28:1–28:4. https://doi.org/10.1145/2642803.2642831
[33] Eelke Folmer and Jan Bosch. 2003. Usability patterns in software architecture. In *Proc. of the 10th Int'l Conf. on Human-Computer Interaction (HCII '03)*. 93–97.
[34] Tak-Chung Fu. 2011. A review on time series data mining. *Eng. Appl. Artif. Intell.* 24, 1 (2011), 164–181. https://doi.org/10.1016/j.engappai.2010.09.007
[35] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empirical Software Engineering* 25, 1 (2020), 678–718. https://doi.org/10.1007/s10664-019-09731-8
[36] David Gilbert. 2020. *JFreeChart*. http://www.jfree.org/jfreechart/
[37] Google. 2020. *Android Studio*. https://developer.android.com/studio
[38] Bernhard Göschlberger and Peter A. Bruck. 2017. Gamification in mobile and workplace integrated microlearning. In *Proceedings of the 19th International Conference on Information Integration and Web-based Applications & Services, iiWAS 2017, Salzburg, Austria, December 4-6, 2017*. 545–552. https://doi.org/10.1145/3151759.3151795
[39] Thomas Green. 2000. Instructions and Descriptions: some cognitive aspects of programming and similar activities. In *Proceedings of the working conference on Advanced visual interfaces, AVI 2000, Palermo, Italy, May 23-26, 2000*. 21–28. https://doi.org/10.1145/345513.345233
[40] Thomas Green and Alan Blackwell. 1998. *Cognitive Dimensions of Information Artefacts: a tutorial*. https://www.cl.cam.ac.uk/~afb21/CognitiveDimensions/CDtutorial.pdf
[41] T. R. G. Green. 1989. Cognitive Dimensions of Notations. In *Proceedings of the Fifth Conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and Computers V*. Cambridge University Press, New York, NY, USA, 443–460. http://dl.acm.org/citation.cfm?id=92968.93015
[42] Ted Hagos. 2019. Android Studio Profiler. In *Android Studio IDE Quick Reference*. Springer, 73–82.
[43] Juho Hamari, Jonna Koivisto, and Harri Sarsa. 2014. Does Gamification Work? - A Literature Review of Empirical Studies on Gamification. In *47th Hawaii International Conference on System Sciences, HICSS 2014, Waikoloa, HI, USA, January 6-9, 2014*. 3025–3034. https://doi.org/10.1109/HICSS.2014.377
[44] Matthias Hauswirth and Trishul M. Chilimbi. 2004. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2004, Boston, MA, USA, October 7-13, 2004*. 156–164. https://doi.org/10.1145/1024393.1024412
[45] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanovic. 2006. Generating object lifetime traces with Merlin. *ACM Trans. Program. Lang. Syst.* 28, 3 (2006), 476–516. https://doi.org/10.1145/1133651.1133654
[46] Trent Hill, James Noble, and John Potter. 2002. Scalable Visualizations of Object-Oriented Systems with Ownership Trees. *J. Vis. Lang. Comput.* 13, 3 (2002), 319–339. https://doi.org/10.1006/jvlc.2002.0238

[47] Andreas Holzinger. 2005. Usability engineering methods for software developers. *Commun. ACM* 48, 1 (2005), 71–74. https://doi.org/10.1145/1039539.1039541

[48] Martin Höst, Björn Regnell, and Claes Wohlin. 2000. Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering* 5, 3 (01 Nov 2000), 201–214.

[49] Michal Hucko, Ladislav Gazo, Peter Simún, Matej Valky, Róbert Móro, Jakub Simko, and Mária Bieliková. 2019. YesElf: Personalized Onboarding for Web Applications. In *Adjunct Publication of the 27th Conference on User Modeling, Adaptation and Personalization, UMAP 2019, Larnaca, Cyprus, June 09-12, 2019*. 39–44. https://doi.org/10.1145/3314183.3324978

[50] Monique W. M. Jaspers, Thiemo Steen, Cor van den Bos, and Maud M. Geenen. 2004. The think aloud method: a guide to user interface design. *I. J. Medical Informatics* 73, 11-12 (2004), 781–795. https://doi.org/10.1016/j.ijmedinf.2004.08.003

[51] JavaMelody. 2020. *JavaMelody : monitoring of JavaEE applications (GitHub).* https://github.com/javamelody/javamelody/wiki

[52] Kamil Jezek and Richard Lipka. 2017. Antipatterns causing memory bloat: A case study. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*. 306–315. https://doi.org/10.1109/SANER.2017.7884631

[53] Zhen Ming Jiang, Ahmed E. Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated performance analysis of load tests. In *25th IEEE International Conference on Software Maintenance (ICSM 2009), September 20-26, 2009, Edmonton, Alberta, Canada*. 125–134. https://doi.org/10.1109/ICSM.2009.5306331

[54] Maria Jump and Kathryn S. McKinley. 2010. Detecting memory leaks in managed languages with Cork. *Softw., Pract. Exper.* 40, 1 (2010), 1–22. https://doi.org/10.1002/spe.945

[55] Reiner Jung and Marc Adolf. 2018. The JPetStore Suite: A concise Experiment Setup for Research. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.

[56] Reiner Jung, Marc Adolf, and Christoph Dornieden. 2017. Towards Extracting Realistic User Behavior Models. *Softwaretechnik-Trends* 37, 3 (2017). http://pi.informatik.uni-siegen.de/stt/37_3/./01_Fachgruppenberichte/SSP2017_proceedings/03_Towards_Extracting_Realistic_User_Behavior_Models.pdf

[57] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*. 327–353. https://doi.org/10.1007/3-540-45337-7_18

[58] Kieker Project. 2013. *Kieker web site.* http://kieker-monitoring.net/

[59] A. J. Ko, Thomas D. LaToza, and Margaret M. Burnett. 2015. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering* 20, 1 (2015), 110–141. https://doi.org/10.1007/s10664-013-9279-3

[60] Lisa Maria Kritzinger, Thomas Krismayer, Rick Rabiser, and Paul Grünbacher. 2019. A User Study on the Usefulness of Visualization Support for Requirements Monitoring. In *7th IEEE Working Conference on Software Visualization*. IEEE, Cleveland, Ohio, USA, 56–66. https://doi.org/10.1109/VISSOFT.2019.00015

[61] Eugene Kuleshov. 2007. Using the ASM framework to implement common Java bytecode transformation patterns. *Aspect-Oriented Software Development* (2007).

[62] Maria Kutar, Carol Britton, and Jonathan Wilson. 2000. Cognitive Dimensions: An experience report. In *Proceedings of the 12th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2000, Cosenza, Italy, April 10-13, 2000*. 7. http://ppig.org/library/paper/cognitive-dimensions-experience-report

[63] Maria Kutar, Chrystopher L. Nehaniv, Carol Britton, and Sara Jones. 2001. The Cognitive Dimensions of an Artifact vis-à-vis Individual Human Users: Studies with Notations for the Temporal Specification of Interactive Systems. In *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings*. 342–355. https://doi.org/10.1007/3-540-44617-6_32

[64] Philipp Lengauer, Verena Bitto, Stefan Fitzek, Markus Weninger, and Hanspeter Mössenböck. 2016. Efficient Memory Traces with Full Pointer Information. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*. 4:1–4:11. https://doi.org/10.1145/2972206.2972220

[65] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2015. Accurate and Efficient Object Tracing for Java Applications. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, Austin, TX, USA, January 31 - February 4, 2015*. 51–62. https://doi.org/10.1145/2668930.2688037

[66] Philipp Lengauer, Verena Bitto, and Hanspeter Mössenböck. 2016. Efficient and Viable Handling of Large Object Traces. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering, ICPE 2016, Delft, The Netherlands, March 12-16, 2016*. 249–260. https://doi.org/10.1145/2851553.2851555

[67] Thomas Lengauer and Robert Endre Tarjan. 1979. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 121–141. https://doi.org/10.1145/357062.357071

[68] Beinan Li, John Ashley Burgoyne, and Ichiro Fujinaga. 2006. Extending Audacity for Audio Annotation. In *ISMIR 2006, 7th International Conference on Music Information Retrieval, Victoria, Canada, 8-12 October 2006, Proceedings*. 379–380.

[69] João Paulo Magalhães and Luís Moura Silva. 2013. Adaptive monitoring of web-based applications: a performance study. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013*. 471–478. https://doi.org/10.1145/2480362.2480454

[70] David Mapelsden, John Hosking, and John Grundy. 2002. Design Pattern Modelling and Instantiation Using DPML. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for Internet, Mobile and Embedded Applications (CRPIT '02)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 3–11. http://dl.acm.org/citation.cfm?id=564092.564094

[71] Mark Marron, César Sánchez, Zhendong Su, and Manuel Fähndrich. 2013. Abstracting Runtime Heaps for Program Understanding. *IEEE Trans. Software Eng.* 39, 6 (2013), 774–786. https://doi.org/10.1109/TSE.2012.69

[72] Karen L. McGraw and Bruce A. McGraw. 1997. Wizards, Coaches, Advisors, and More: A Performance Support Primer. In *Human Factors in Computing Systems, CHI '97: Looking to the Future, Extended Abstracts, Atlanta, Georgia, USA, March 22-27, 1997*. 152–153. https://doi.org/10.1145/1120212.1120318

[73] Nick Mitchell. 2006. The Runtime Structure of Object Ownership. In *ECOOP 2006 - Object-Oriented Programming, 20th European Conference, Nantes, France, July 3-7, 2006, Proceedings*. 74–98. https://doi.org/10.1007/11785477_5

[74] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making Sense of Large Heaps. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*. 77–97. https://doi.org/10.1007/978-3-642-03013-0_5

[75] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2010. Four Trends Leading to Java Runtime Bloat. *IEEE Software* 27, 1 (2010), 56–63. https://doi.org/10.1109/MS.2010.7

[76] Nick Mitchell and Gary Sevitsky. 2003. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings*. 351–377. https://doi.org/10.1007/978-3-540-45070-2_16

[77] Nick Mitchell and Gary Sevitsky. 2007. The causes of bloat, the limits of health. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*. 245–260. https://doi.org/10.1145/1297027.1297046

[78] MyBatis. 2016. *JPetStore*. http://mybatis.org/jpetstore-6/

[79] Nicholas Nethercote and Julian Seward. 2003. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.* 89, 2 (2003), 44–66. https://doi.org/10.1016/S1571-0661(04)81042-9

[80] Jakob Nielsen. 1993. *Usability engineering*. Academic Press.

[81] Mie Nørgaard and Kasper Hornbæk. 2006. What do usability evaluators do in practice?: an explorative study of think-aloud testing. In *Proceedings of the Conference on Designing Interactive Systems,University Park, PA, USA, June 26-28, 2006*. 209–218. https://doi.org/10.1145/1142405.1142439

[82] Oracle. 2014. *Java Flight Recorder*. https://docs.oracle.com/javacomponents/jmc-5-4/jfr-runtime-guide/about.htm#JFRUH170

[83] Oracle. 2018. *JConsole*. https://docs.oracle.com/javase/7/docs/technotes/guides/management/jconsole.html

[84] Oracle. 2018. *jmap*. https://docs.oracle.com/javase/7/docs/technotes/tools/share/jmap.html

[85] Oracle. 2020. *HPROF: A Heap/CPU Profiling Tool*. https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html

[86] Oracle. 2020. *Java Mission Control*. https://openjdk.java.net/projects/jmc/

[87] Oracle. 2020. *JVM Tool Interface Version 1.2*. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html

[88] Kelly O'Hair. 2004. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network, Developer Technical Articles & Tips* 28 (2004).

[89] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proc. of the 40th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2019)*.

[90] Rick Rabiser, Paul Grünbacher, and Martin Lehofer. 2012. A qualitative study on user guidance capabilities in product configuration tools. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*. 110–119. https://doi.org/10.1145/2351676.2351693

[91] Rick Rabiser, Michael Vierhauser, and Paul Grünbacher. 2016. Assessing the usefulness of a requirements monitoring tool: a study involving industrial software engineers. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 122–131. https://doi.org/10.1145/2889160.2889234

[92] Derek Rayside and Lucy Mendel. 2007. Object ownership profiling: a technique for finding and fixing memory leaks. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*. 194–203. https://doi.org/10.1145/1321631.1321661

[93] Steven P. Reiss. 2009. Visualizing the Java heap to detect memory problems. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2009, Edmonton, Alberta, Canada, September 25, 2009*. 73–80. https://doi.org/10.1109/VISSOF.2009.5336418

[94] Jan Renz, Thomas Staubitz, Jaqueline Pollack, and Christoph Meinel. 2014. Improving the Onboarding User Experience in MOOCs. In *Proc. of the 6th Int'l Conf. on Education and New Learning Technologies (EDULEARN '14)*.

[95] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2011. Elephant Tracks: generating program traces with object death records. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ 2011, Kongens Lyngby, Denmark, August 24-26, 2011*. ACM, 139–142. https://doi.org/10.1145/2093157.2093178

[96] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. 2013. Elephant tracks: portable production of complete and precise gc traces. In *International Symposium on Memory Management, ISMM 2013, Seattle, WA, USA, June 20, 2013*. ACM, 109–118. https://doi.org/10.1145/2491894.2466484

[97] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering* 14, 2 (2009), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[98] Anthony Savidis and Nikos Koutsopoulos. 2011. Interactive Object Graphs for Debuggers with Improved Visualization, Inspection and Configuration Features. In *Advances in Visual Computing - 7th International Symposium, ISVC 2011, Las Vegas, NV, USA, September 26-28, 2011. Proceedings, Part I*. 259–268. https://doi.org/10.1007/978-3-642-24028-7_24

[99] Roger C Schank, Tamara R Berman, and Kimberli A Macpherson. 1999. Learning by doing. *Instructional-design theories and models: A new paradigm of instructional theory* 2, 2 (1999), 161–181.

[100] Carolyn B. Seaman. 1999. Qualitative Methods in Empirical Studies of Software Engineering. *IEEE Trans. Software Eng.* 25, 4 (1999), 557–572. https://doi.org/10.1109/32.799955

[101] Ben Shneiderman. 1996. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages, Boulder, Colorado, USA, September 3-6, 1996*. IEEE Computer Society, 336–343. https://doi.org/10.1109/VL.1996.545307

[102] Connie U. Smith and Lloyd G. Williams. 2000. Software performance antipatterns. In *Second International Workshop on Software and Performance, WOSP 2000, Ottawa, Canada, September 17-20, 2000*. 127–136. https://doi.org/10.1145/350391.350420

[103] Ken Soong, Xin Fu, and Yang Zhou. 2018. Optimizing New User Experience in Online Services. In *5th IEEE International Conference on Data Science and Advanced Analytics, DSAA 2018, Turin, Italy, October 1-3, 2018*. 442–449. https://doi.org/10.1109/DSAA.2018.00057

[104] Vladimir Sor, Plumbr Ou, Tarvo Treier, and Satish Narayana Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013*. 544–547. https://doi.org/10.1109/ICSM.2013.92

[105] Vladimir Sor and Satish Narayana Srirama. 2014. Memory leak detection in Java: Taxonomy and classification of approaches. *Journal of Systems and Software* 96 (2014), 139–151. https://doi.org/10.1016/j.jss.2014.06.005

[106] Eclipse Foundation. 2020. *Eclipse Memory Analyzer (MAT)*. https://www.eclipse.org/mat/

[107] Oracle. 2020. *The HotSpot Group*. http://openjdk.java.net/groups/hotspot/

[108] Oracle. 2020. *VisualVM: All-in-One Java Troubleshooting Tool*. https://visualvm.github.io/

[109] Doug Tidwell and Jeanette Fuccella. 1997. TaskGuides: Instant Wizards on the Web. In *The 15th Annual International Conference of Computer Documentation: Crossroads in Communication, SIGDOC 1997, Salt Lake City, Utah, USA, October 19-22, 1997*. 263–272. https://doi.org/10.1145/263367.263401

[110] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. 2009. *Continuous Monitoring of Software Services: Design and Application of the Kieker Framework*. Technical Report TR-0921. Department of Computer Science, Kiel University, Germany. 27 pages pages.

[111] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. 2012. Kieker: a framework for application performance monitoring and dynamic software analysis. In *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE'12, Boston, MA, USA - April 22 - 25, 2012*. 247–248. https://doi.org/10.1145/2188286.2188326

[112] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011*. 213–222. https://doi.org/10.1109/ICSM.2011.6080788

[113] Markus Weninger et al. 2020. *AntTracks - Memory Monitoring using Accurate and Efficient Object Tracing for Java Applications*. http://mevss.jku.at/AntTracks

[114] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Utilizing object reference graphs and garbage collection roots to detect memory leaks in offline memory monitoring. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes, ManLang 2018, Linz, Austria, September 12-14, 2018*. 14:1–14:13. https://doi.org/10.1145/3237009.3237023

[115] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Analyzing Data Structure Growth Over Time to Facilitate Memory Leak Detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 7-11, 2019*. 273–284. https://doi.org/10.1145/3297663.3310297

[116] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2019. Detection of suspicious time windows in memory monitoring. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2019, Athens, Greece, October 21-22, 2019*. 95–104. https://doi.org/10.1145/3357390.3361025

[117] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. 2018. Analyzing the Evolution of Data Structures Over Time in Trace-Based Offline Memory Monitoring. In *Proc. of the 9th Symposium on Software Performance (SSP '18)*.

[118] Markus Weninger, Paul Grünbacher, Huihui Zhang, Tao Yue, and Shaukat Ali. 2018. Tool Support for Restricted Use Case Specification: Findings from a Controlled Experiment. In *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. 21–30. https://doi.org/10.1109/APSEC.2018.00016

[119] Markus Weninger, Philipp Lengauer, and Hanspeter Mössenböck. 2017. User-centered Offline Analysis of Memory Monitoring Data. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*. 357–360. https://doi.org/10.1145/3030207.3030236

[120] Markus Weninger, Lukas Makor, Elias Gander, and Hanspeter Mössenböck. 2019. AntTracks TrendViz: Configurable Heap Memory Visualization Over Time. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE 2019, Mumbai, India, April 07-11, 2019*. 29–32. https://doi.org/10.1145/3302541.3313100

[121] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. 2019. Memory Leak Visualization using Evolving Software Cities. In *Proc. of the 10th Symposium on Software Performance (SSP '19)*.

[122] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018, Berlin, Germany, April 09-13, 2018*. 115–126. https://doi.org/10.1145/3184407.3184412

[123] Nicholas P. Wilde. 1996. Using Cognitive Dimensions in the Classroom as a Discussion Tool for Visual Language Design. In *Conference on Human Factors in Computing Systems: Common Ground, CHI '96, Vancouver, BC, Canada, April 13-18, 1996, Conference Companion*. ACM, 187–188. https://doi.org/10.1145/257089.257252

[124] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing. *ACM Trans. Softw. Eng. Methodol.* 23, 3 (2014), 23:1–23:50. https://doi.org/10.1145/2560047

[125] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*. 421–426. https://doi.org/10.1145/1882362.1882448

[126] Guoqing (Harry) Xu and Atanas Rountev. 2008. Precise memory leak detection for java software using container profiling. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*. 151–160. https://doi.org/10.1145/1368088.1368110

[127] Guoqing (Harry) Xu and Atanas Rountev. 2010. Detecting inefficiently-used containers to avoid bloat. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*. 160–173. https://doi.org/10.1145/1806596.1806616

[128] S. Zaman, B. Adams, and A. E. Hassan. 2012. A Large Scale Empirical Study on User-Centric Performance Analysis. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 410–419. https://doi.org/10.1109/ICST.2012.121

[129] Thomas Zimmermann and Andreas Zeller. 2001. Visualizing Memory Graphs. In *Software Visualization, International Seminar Dagstuhl Castle, Germany, May 20-25, 2001, Revised Lectures*. 191–204. https://doi.org/10.1007/3-540-45875-1_15

Received October 25th, 2019; revised December 12th, 2019; accepted January 11th, 2020