Visualizing Memory Consumption with Vismep

Alison Fernandez Blanco*, Alexandre Bergel‡, Juan Pablo Sandoval Alcocer†, Araceli Queirolo Córdova*
*DCC, University of Chile
‡RelationalAI, Switzerland – https://bergel.eu
†Department of Computer Science, School of Engineering, Pontificia Universidad Católica de Chile

***Abstract*—Detecting and repairing memory issues is still a challenging task. One reason is that understanding a program's memory usage involves a diverse and related set of dynamic and static aspects. Over the years, multiple tools have been proposed to assist practitioners in these activities. However, detailed information about how a tool helps users when analyzing memory usage is missing.**

This article introduces Vismep, an interactive visualization prototype to help programmers analyze Python applications' memory usage, and presents an exploratory study to understand the behavior and perception of users when using Vismep. As a result, we reported five information needs when participants analyze memory consumption and how they use Vismep to satisfy these needs. Besides, participants positively perceived Vismep due to their valuable views and high overall usability.

I. Introduction

Monitoring and understanding an application's memory usage help programmers discover anomalies. Such anomalies may be non-trivial and possible outcomes include crashes and performance degradation [1], [2], [3]. To assist programmers in identifying memory anomalies and evaluate how well programs perform based on multiple aspects (*e.g.,* memory usage, memory allocations, memory access), numerous memory profiling tools have been proposed [4], [5], [6]. Typically, these tools report the information related to memory usage through full-text reports or tables. Moreover, several software visualizations with interaction mechanisms were considered suitable alternatives for helping programmers in examining and addressing memory issues [7], [8], [9], [10] since visualizations are known to be adequate at supporting practitioners in software comprehension [11], [12].

Although various tools are provided, programmers usually need substantial time to understand memory usage, detect anomalies, and repair issues [13], [14]. For example, it has been reported that users employ different tools to address memory issues in popular Python packages (*e.g.,* pandas[1], scikit-learn[2]). However, the discussions indicate that users struggle to understand the information provided by those tools. Furthermore, little is known about what information programmers need when analyzing memory and how programmers employ tools to find this information [22].

This paper introduces Vismep, an interactive visualization prototype to help programmers analyze Python application memory usage. Vismep gathers and reports information (*e.g.,* calls between functions, memory usage) using polymetric views [15]. In order to explore and analyze the behavior and perception of programmers when employing Vismep, we conducted an exploratory study with eleven participants who analyzed their software's memory consumption using Vismep. We carefully monitored which information programmers usually look for when analyzing memory usage, how they use Vismep to obtain this information, and how they perceive Vismep.

Our findings indicate that programmers look for dynamic and static information to (a) identify relevant code - the functions/methods involved in implementing specific behavior or belonging to particular modules, (b) locate allocation hotspots - the functions/methods or code lines that allocate most memory, (c) inspect circumstances, rationale, and events of selected functions/methods - the circumstances in which functions/methods are executed, their rationale and the memory events (allocation, access, release) related, (d) detect memory anomalies - code involved with excessive or inefficient memory usage, and (e) trace the cause of anomalies - how memory anomalies affect memory usage behavior. Additionally, programmers used a wide range of Vismep features to perform previously mentioned activities. Furthermore, we found missing information and opportunities for improvement that could guide the design and implementation of Vismep and other tools. We also noticed that Vismep is positively perceived since participants indicated a low to moderate mental workload effort when using it and considered that Vismep offers high overall usability.

Contributions. In summary, this paper makes the following contributions:

- An interactive visualization prototype, Vismep that supports programmers in analyzing memory consumption over Python applications.
- The information needs - relevant code, allocation hotspots, circumstances, rationale, and events, memory anomalies, and cause of anomalies - that programmers have when analyzing memory consumption.
- A detailed report that summarizes how programmers employ Vismep to obtain the required information and how programmers perceive Vismep.

Paper structure. Section II summarizes the prior work. In Section III, we describe in detail Vismep. Section IV presents the methodology to evaluate Vismep. Section V describes the results. Section VI discusses the threats to validity. Finally, Section VI concludes and outlines future work.

[1]https://github.com/pandas-dev/pandas/pull/45489
[2]https://github.com/scikit-learn/scikit-learn/issues/19774

II. Prior Work

This section highlights some of the most relevant prior works.

Software visualization for memory usage analysis. Software visualization employs multiple techniques to display a variety of information and support programmers when analyzing memory usage. Several studies show the calls between functions/methods with a memory footprint (*e.g.,* memory allocations, memory accesses, memory releases) using node-link diagrams or stacked displays [8], [9], [10], [16], [17], [18]. These visualizations help programmers explore and locate functions/methods related to problematic memory events (allocations, accesses, releases), leading to memory anomalies (*e.g.,* memory bloat, memory leak). Other studies propose visualizations to represent references between objects and assist programmers in memory leak detection by highlighting objects not reclaimed by the garbage collector [16], [19], [20]. Also, showing this information is helpful for data structure analysis by locating objects shared by data structures [21]. Although Vismep does not introduce a novel visualization technique, it adequately combines demonstrated techniques. Also, it connects the source code with information from program execution, something that most visualizations dismiss [22]. Furthermore, Vismep supports visualizations of Python applications, which despite Python's popularity, it is not often supported by proposed visualizations.

Software visualization evaluation. Software visualizations that assist programmers with memory consumption analysis are difficult to evaluate. Most studies usually evaluate these software visualization approaches through usage scenarios [8], [10], [17], [23], [24], showing the benefits of approaches. Nevertheless, little is known about how diverse programmers use and perceive visualizations to inspect applications' memory usage. According to Fernandez *et al.* [22], evaluating visualizations with developers could be challenging since it may require participants experienced in memory monitoring. In addition, providing detailed evidence as to whether an approach is adequate to support some programmers' needs in this problem domain is problematic since these needs are not thoroughly researched yet. Consequently, few software visualizations are evaluated through user studies [9], [16], [25]. Among them, most focus on task completion and correctness. We consider analyzing other variables to recognize the effect that visualizations have in this context. Therefore, we (i) identified the information required by programmers when analyzing memory usage using Vismep, (ii) explored how programmers employ Vismep to obtain this information, and (iii) analyzed how programmers perceive Vismep.

III. Vismep

Vismep is an interactive visualization prototype designed to help programmers in analyzing the memory usage of Python applications. Vismep collects memory traces during program execution and displays the information through four complementary views.

Vismep profiler. To extract multiple aspects of the program execution, we designed a profiler for Vismep. This profiler is based on two popular python modules, *memory_profiler*[3] and *trace*[4]. The Vismep profiler collects the following data:

- A set of invoked functions/methods with detailed information (name, file, number of lines of code, number of executions) during program execution.
- A memory footprint (memory consumed, memory used per code line) for each invoked function/method.
- The calling relationships between invoked functions/methods.

After gathering the data with Vismep profiler, we employed polymetric views [12] to illustrate the information obtained. Vismep implementation and the learning material are available[5].

A. In a Nutshell

To illustrate Vismep, we analyzed a memory issue reported in the pandas package[6]. Pandas is a flexible and powerful package for supporting programmers in data science/data analysis and machine learning tasks. The issue reported was reproducible with the piece of code illustrated at the right of Figure 1 (SC). The code essentially creates a dictionary (`data`) in line 2, and between lines 3 and 5, a `dataframe` object is created based on `data` and converts it into a JSON string several times.

Figure 1 gives an overview of Vismep. The left view (CG) is the *Call graph view* that displays the calling relationships between the invoked functions/methods. In the *Call graph view*, the `export_dataframe` function that consumes 728.474 MB is selected. Consequently, the *Source code view* (SC) is displayed on the right-hand side to show the source code of the selected function (`export_dataframe`). Figure 4 and Figure 3 illustrate the alternative views of *Call graph view* and *Source code view* respectively. Figure 4 shows the *Scatter plot view* that presents a graph to assist the user in quickly noting the relationship between the memory consumed and the number of executions of the functions/methods invoked. Figure 3 displays the *Sub call graph view* that indicates the callers and callees of a particular function, in this case the `to_json` function.

B. Call Graph View

This view helps users understand how the program runs and uses memory. It shows a node-link diagram commonly used to illustrate the calling relationships between functions/methods [22]. It also displays the memory footprint and additional information (*e.g.,* name, number of executions, size) for each executed function/method.

Nodes. As Figure 2 illustrates, each node is a function/method invoked during the program execution. The visual mapping of a node is the following:

- The width represents the average memory consumed (MB) by a function/method.

[3]https://github.com/pythonprofilers/memory_profiler
[4]https://github.com/python/cpython/blob/3.10/Lib/trace.py
[5]https://github.com/Balison/Vismep
[6]https://github.com/pandas-dev/pandas/pull/45489

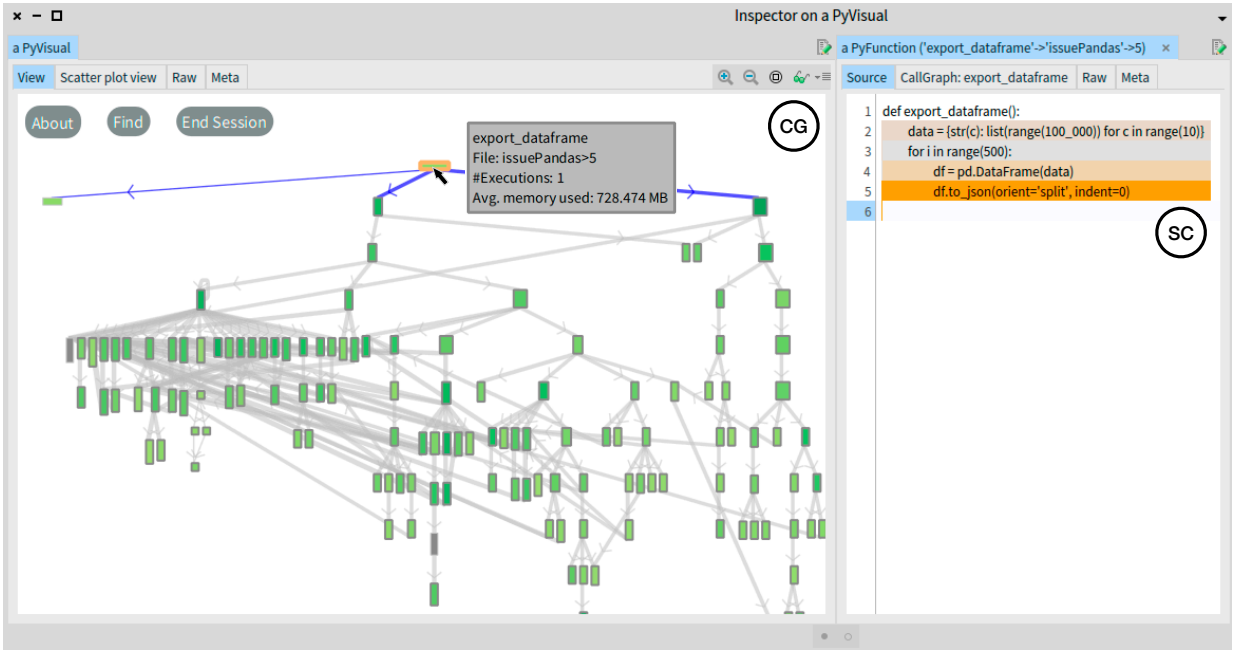

Fig. 1: Illustrating a memory issue of pandas with Vismep. *Call Graph view* (CG) shows the memory usage along the execution path. Each node denotes an executed function, and the edges indicate the calling relationships. The width node shows a function's average memory, and the height denotes the times a function is executed. When a function is selected, its border turns orange, and its source code is displayed in *Source Code view* (SC). Each line background from the source code denotes the memory usage increment from gray (low increase) to orange (high increase). If there is no increase, the background is white.

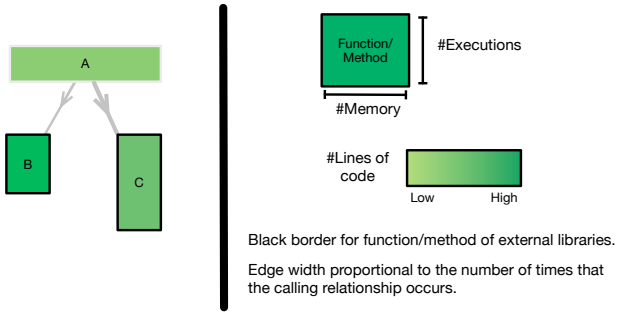

Fig. 2: On the right, an example with *Call graph view*, where `A` function is executed few times and consumes a lot of memory. `A` calls first `B` (a few times) and then `C` (several times). On the left, the visual mapping for *Call graph view* and *Sub call graph view*.

- The height denotes how many times the function/method is executed.
- The color indicates the number of lines of code used to define the function/method. The color varies from light to dark green, the darker the node the greater the number of lines of code. However if the source code (*e.g.,* defined in native C sources) cannot be retrieved by inspect package[7], the color is gray.
- The border shows if the function/method belongs to an external library (*e.g.,* pandas, random).

[7]https://docs.python.org/3/library/inspect.html

For instance, `export_dataframe` function is the widest node and has the least height of the nodes in the view since it consumes around 728 MB with a single execution in Figure 1.

Edges. Edges between functions/methods indicate the calling relationships. The edge's arrow indicates the direction of the calling relationship to help users distinguish a caller from a callee. The edge's width denotes the number of times the calling relationship occurs during program execution. For example, Figure 2 shows that `A` function calls to `B` function and `C` function. Also, displays that `A` function calls more times to `C` function than to `B` function, due to the width of edges.

Layout. The functions/methods are located in the view using a vertical tree layout. As a result, the roots that usually include `main` function are located at the top, and leaves are placed at the bottom. Additionally, the functions/methods are sorted based on the invocation order from right to left.

C. Source Code View

When a function/method is selected, a *Source code view* is built at the right, as shown in Figure 1 (SC). This view displays the source code of the selected function/method and highlights the background code lines based on the memory used. The background fades from light gray (*i.e.,* little memory usage) to orange (*i.e.,* high memory usage) depending on how much memory consumption increased after executing that line. A white background indicates that the memory did not increase. Consequently, this view connects dynamic aspects

with source code, so users can easily identify the code piece that allocates most memory or anomalies [22]. To illustrate, in Figure 1, we observe that line 5 allocates the highest amount of memory when `df` is converted to a JSON string, thus, `export_dataframe` function calls to `to_json` function.

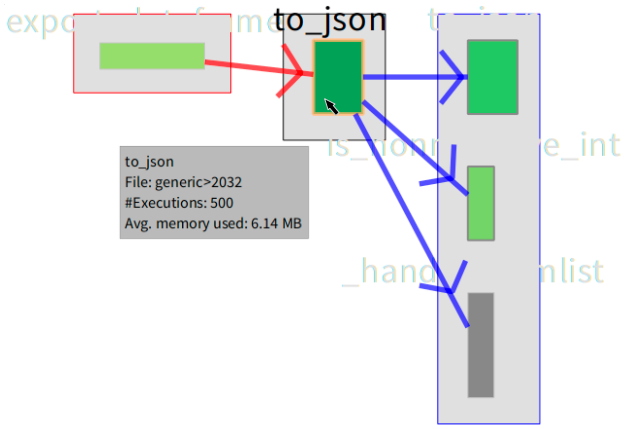

Fig. 3: Overview of Sub call graph view.

D. Sub Call Graph View

Vismep assists the user in quickly identifying the execution path of a particular node that she/he would like to investigate. When selecting the *Callgraph* tab at the top of the *Source code view*, the *Sub call graph view* is shown instead of the source code. Figure 3 illustrates the *Sub call graph view* that presents a summarized call graph based on a function/method. It visualizes the callers and callees of a selected function/method, where the callers and callees are located at the left and right of the selected function/method, respectively. For example, we can observe in Figure 3 that `to_json` function is called by `export_dataframe` function 500 times. For each time that `to_json` function is called, `to_json` function calls to other three functions: `to_json`, `is_nonnegative_int`, and `_handle_fromlist` (part of a C library). Node labels are placed *behind* nodes to not clutter the visualization. The effect is to favor an unobstructed layout of nodes. A label is temporarily moved to the foreground when the mouse hovers a node, as illustrated in Figure 3.

E. Scatter Plot View

Vismep supports users in determining the relationship between the total memory consumed (sum of memory allocated per execution) and the number of executions of the functions/methods invoked. *Scatter plot view* represents each function/method as a point (e, m) where e is the number of times the function/method is executed (X-axis), and m the amount of memory allocated by the function/method (Y-axis). The color of each point ranges from light to intense green to indicate the size in terms of lines of code. Figure 4 shows that `_write` function allocates most total memory (around 3077 MB) and it is executed 500 times. This function is called indirectly in line 5 at the `export_dataframe` function (Figure 1), and focuses on converting the `df` to a JSON string. More specifically, `_write` function allocates around 6 MB each time it is executed. Figure 4 also illustrates some functions/methods in which the total memory used is negative. The latter indicates that during some execution or executions of a function/method, the garbage collector is activated, and several blocks of memory are released. Functions releasing memory have a negative memory allocation.

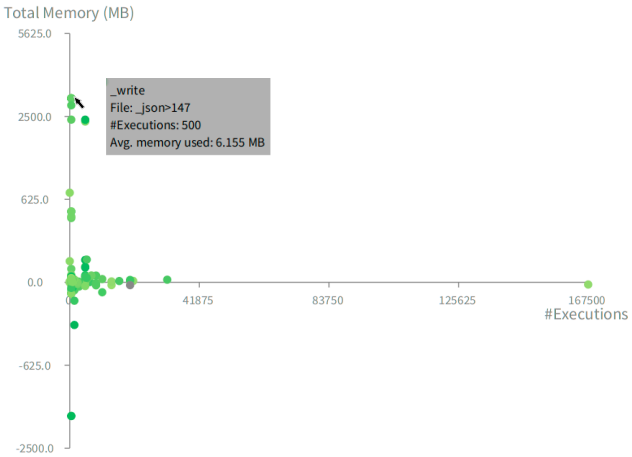

Fig. 4: Overview of Scatter plot view.

F. Interactions

Vismep provides a number of interactions to facilitate the exploration of the Python application under analysis.

Canvas movement. The user can pan the view around the different visualizations, zoom in and out by arbitrary distances, and zoom the display to fit the entire visualization.

Mouse hovering. As Figure 1 and Figure 3 shows, when the user hovers the mouse cursor above an invoked function/method, a popup window appears with information about the respective function/method, such as the name, the number of executions, and the average amount of memory consumed. If the user performs this action over an invoked function/method in the *Call graph view* (Figure 1) and the *Sub call graph view* (Figure 3), the incoming and outgoing edges are highlighted in red and blue, respectively.

Drag. The user can select a function/method node and drag the node with all its callee nodes to change the position of nodes in the *Call graph view* and *Sub call graph view*. Manually dragging it is useful to cluster nodes in an ad-hoc fashion.

Search. Vismep offers the user a search button at the top of the *Call graph view* (Figure 1) that performs a search over the name of an invoked function/method. The user should select the desired function/method from a window that enlists the functions/methods that fulfill the query. Consequently, the selected function/method is highlighted.

Drill down. Vismep provides the user an option to obtain detailed data about a particular invoked function/method. Clicking a function/method shows two views: *Source code view* (Figure 1) and *Sub call graph view* (Figure 3).

IV. Methodology

We designed and conducted an exploratory study to understand how programmers employ Vismep when analyzing memory usage of Python applications. The following subsections explain the steps of our study.

A. Research Questions

Our study is designed to answer the following research questions (RQ):

- **RQ1:** How does Vismep support programmers when analyzing memory consumption?
  - **RQ1.1:** What information do programmers look for when analyzing memory consumption using Vismep?
  - **RQ1.2:** How do programmers employ Vismep to obtain this information?
- **RQ2:** How do programmers perceive Vismep when analyzing memory consumption?
  - **RQ2.1:** How does Vismep impact the cognitive load?
  - **RQ2.2:** How useful do programmers consider Vismep?
  - **RQ2.3:** What are the perceptions of the programmers on the current features of Vismep?

To respond RQ1, we examined the behavior of programmers and the actions made with Vismep during memory consumption analysis. To answer RQ2, we collected impressions of the cognitive load and the usability perceived by programmers when employing Vismep. We also extracted the participant's feedback from the features offered by Vismep.

B. Participants and Applications

We invited students and bachelors from our university and members of Python communities to participate in our study.

We selected eleven programmers who freely opted to participate, all familiar with Python programming. Their average age was 27 years old (std. dev. 1.9). Participants were from diverse fields of study; three participants have or are pursuing a degree in Computer Science, and the remaining in other fields (*e.g.,* Geology, Electrical). Two participants were from the industry, three were in research centers, three pursued a master's degree, and the rest were bachelors.

Participants exhibited various levels of experience in Python programming. Their average experience in Python was 4.6 years (std. dev. 2.5). Participants also self-assessed their expertise using a Likert scale of five steps *i.e.,* 1 (no experience) to 5 (expert). The average experience in Python programming was 3.4 (std. dev. 0.8).

Experience in memory usage analysis. Eight participants showed experience examining memory usage and addressing memory issues. We asked experienced participants how they usually monitor memory consumption or manage memory anomalies in Python applications, and we detected some strategies used:

- *Manual.* Five participants usually trace the code execution to identify a piece of code (*e.g.,* unused data, allocation sites) with the risk of causing memory anomalies (*e.g.,* memory bloat, memory leak).
- *Logs.* Two participants usually insert events (*e.g.,* print messages) at the functions or methods they consider prone to memory issues. For instance, they print a message when a specific data structure is created, modified or accessed.
- *Web search.* One participant prefers to perform a web search with the characteristics involved with a memory issue to repair it.

Projects under study. We described explicitly in the invitation that the study focused on understanding how programmers analyze memory consumption in Python applications. We also specified that volunteers who decide to participate in this study must themselves choose a Python application that they find interesting to examine in terms of memory usage. Consequently, participants selected different programs (*e.g.,* data analysis, IA, ML) that they thought might be interesting in terms of memory consumption and with which they were familiar.

C. Procedure

The study consisted of carrying out a work session for each participant with her/his selected application. A work session begins with the moderator presenting the objective and the characteristics of the study described in the invitation to programmers who agreed to participate. The moderator also asked the participant to use the think-aloud technique [26] during the session.

Additionally, each work session is structured as follows:

1) *Background and expectations.* The participant answered general questions to gather demographic data such as their age, gender, level of experience in Python programming, and addressing memory issues. The moderator then asked the participant for an opinion about the memory consumed by her/his application. The participant also explains which elements (*e.g.,* functions, methods, allocations) may produce a memory anomaly (*e.g.,* memory bloat, memory leak) during program execution.
2) *Exploration.* The participant reads the learning material of Vismep and had an exploration phase to familiarize herself/himself with the visualization tool.
3) *Tasks.* The participant employed Vismep to analyze the memory usage of her/his application and responds to the questions listed in Table I.
4) *Online forms.* The participant filled out two online forms to measure the mental workload (NASA-TLX) [27] and the perceived usability of Vismep (SUS) [28]. These two self-assessments techniques are hugely popular in empirical studies and are applicable in our case.
5) *Post-study questionnaire.* Finally, the participant answered verbally and informally open questions regarding their observations, perceptions, and desired improvements of Vismep.

We observed, tracked, and monitored the participants' interactions with Vismep throughout work sessions. We also recorded a video of the screen and the audio of the laptop used by the participants.

Category	Question	Rationale
Characterizing memory usage	**Q1:** *Can you characterize the memory consumption of your application?*	The participant identifies and describes the information relevant to the memory usage analysis (*e.g.,* allocation sites, allocations made).
Understanding memory usage	**Q2:** *What have you learned from your application? Do you find anything surprising (*e.g., *anomalies)?*	The participant contrasts the information provided by Vismep with her/his assumptions. Also, she/he explains if Vismep provides additional and unknown information and which potential issues may exist in her/his program.
Optimizing memory usage	**Q3:** *Do you find an opportunity to decrease memory consumption?* **Q4:** *If you find an opportunity to decrease memory usage, can you improve it and run the profiler again?*	The participant localizes and explains which parts of the code may be modified to reduce the memory usage of her/his program. The participant modifies the code's parts that are assumed to be the root cause of a memory anomaly. Also, she/he employs Vismep over the changed program to verify the impact of the changes in memory usage.

TABLE I: Questions made to the participant.

D. Data Collection

We collected a variety of data to answer our research questions. Next, we discuss the data collection process.

Interactions extraction. To answer RQ1, we collected the actions made by participants when using Vismep to answer the questions in Table I. We reviewed and checked the tracking logs and video recordings from sessions to generate spreadsheets that summarize the work sessions. Each spreadsheet presents (i) the question asked by the moderator, (ii) the verbalized thoughts of participants, (iii) the corresponding period of time in the video records, (iv) the actions made by the participants, and (v) the Vismep views used. To minimize biases during this process, one author generated the spreadsheets, and another author checked if the data was consistent with the audio, video records, and tracking logs.

User experience extraction. To respond to RQ2, we gathered the answers from the following online forms:

- *NASA-TLX.* It is widely used to measure subjective mental workload [27]. NASA-TLX derives an overall workload score based on six workload dimensions: mental demand, physical demand, temporal demand, performance, effort, and level of frustration.
- *SUS.* The System Usability Scale is a reliable and standard technique to evaluate the usability of a system [28]. This questionnaire contains ten statements to measure the perceived usability of a system.

Next, we transferred the results of NASA-TLX and SUS questionnaires into a spreadsheet for computational purpose. We also collected the responses of the participants corresponding to the post-study questionnaire. Besides, we checked the verbalized thoughts of participants to extract information related to the perception of Vismep features.

E. Data Analysis

This section illustrates the methods used to analyze the gathered data.

Interactions analysis. One author analyzed the spreadsheets using open and descriptive coding [29] to identify themes (activities) related to the information required by participants, similar to the study of Velez and colleagues [30] (RQ1.1). Another author checked the consistency of codes and observations to minimize biases during this process. Next, based on the generated codes, an author checked which actions were performed to obtain the data required using Vismep (RQ1.2).

User experience analysis. We calculated and examined the NASA-TLX and SUS scores reported by eleven participants to respond to RQ2.1 and RQ2.2. To answer RQ2.3, we followed the open coding method [31] to analyze the answers to the post-study questionnaire. First, we detected concepts and keywords in the collected data. We then grouped the concepts to generate coherent groups (categories) that highlight broader patterns.

V. Results

This section details the results to answer the two research questions proposed in the study.

A. RQ1.1: Information needs

We identified that participants looked for dynamic and static information for five themes to answer questions (Table I). Table II lists the themes and the number of participants that looked for each theme. We next detailed what information participants searched to respond to the questions based on the following categories.

Characterizing memory usage. When participants characterized the memory used in their applications, all of them identified relevant code; the functions/methods considered vital for the program functionality based on participants' knowledge about the program under analysis. Also, all participants searched for allocation hotspots; the functions/methods or code lines that allocate most memory. After participants detected functions/methods from relevant code or allocation hotspots, they often expanded their information by understanding their circumstances, rationale, and events. More specifically, as the participants knew which functions/methods were interesting to them (relevant code, allocation hotspots), participants wanted detailed information about (i) the circumstances that caused their execution, (ii) the intention behind their implementation (*i.e.,* rationale), and (iii) the memory events (allocations, accesses, releases) related with them.

Understanding memory usage. To understand the memory used by their applications, participants located relevant code and allocation hotspots. Participants then contrasted the memory consumed by those functions/methods with the assumptions that participants held.

Theme	Information need for	Freq.	Actions	Explored views	Freq.
Relevant code	Detect code that is involved in implementing certain behavior or belonging to particular modules, files.	11/11	Search functions/methods based on name. Search functions/methods based on module.	Call graph view	11/11
			Inspect the functions/methods source code.	Source code view	4/11
Allocation hotspots	Detect code that allocates most memory.	11/11	Discover and compare memory usage of functions/methods.	Call graph view Scatter plot view Sub call graph view	11/11 9/11 2/11
			Discover and compare memory usage of code lines.	Source code view	11/11
Circumstances, rationale and events	Understand under what circumstances functions/methods are executed, their rationale and the memory events related.	11/11	Inspect the functions/methods callers, callees and execution path.	Call graph view Sub call graph view	11/11 7/11
			Inspect functions/methods rationale. Inspect functions/methods memory events (allocations, accesses, releases)	Source code view	11/11
Memory anomalies	Locate code involved with excessive or inefficient memory usage.	11/11	Analyze memory usage of allocation hotspots or relevant code.	Call graph view Scatter plot view	11/11 3/11
			Inspect the circumstances, rationale and events of allocation hotspots or relevant code.	Call graph view Source code view	7/11 5/11
Anomalies cause	Locate the root cause of an anomaly.	7/11	Inspect the circumstances, rationale and events of memory anomalies. Analyze how memory anomalies affect the memory usage and functionality of relevant code	Call graph view Source code view	7/11 4/11

TABLE II: Information needs, actions, and views explored in Vismep to analyze Python applications.

Participants also tried to locate memory anomalies; code involved with excessive or inefficient memory usage. When detecting memory anomalies, participants mostly attempted to identify unexpected memory usage behavior in relevant code and allocation hotspots and determine if the memory consumed was necessary or not for the proper functionality of the program.

Optimizing memory usage. When participants tried to reduce the memory consumed by their application, most traced the anomalies cause; the root cause responsible for excessive or inefficient memory usage (memory anomalies). Then, these participants analyzed how to address the memory anomalies based on the anomalies' cause.

Other participants considered that their applications do not contain a memory anomaly. Consequently, they located allocation hotspots and tried to comprehend their circumstances, rationale, and events for locating an optimization opportunity.

RQ1.1: Programmers looked for dynamic and static information to (a) identify relevant code; code involved in implementing certain behavior or that belongs to particular modules, (b) locate allocation hotspots; code that allocate most memory, (c) inspect circumstances, rationale, and events of selected functions/methods; the circumstances in which functions/methods are executed, their rationale and the memory events (allocation, access, release) related, (d) detect memory anomalies; code involved with excessive or inefficient memory usage, and (e) trace the cause of anomalies; how memory anomalies affect memory usage behavior.

B. RQ1.2: Use of Vismep

Table II lists the actions that participants performed to look for each theme, the Vismep views used to execute the actions, and the number of participants that performed those actions per view. We described how participants employed Vismep to get the information needed for the identified themes in the following.

Characterizing memory usage. All the participants characterized the memory used by their application using Vismep. Consequently, participants located relevant code, detected allocation hotspots and explored the circumstances, rationale, and events of functions/methods of interest.

When looking for relevant code, participants searched in the *Call graph view* functions/methods based on their name or the module to which they belong. Some participants were unsure about the rationale behind a function/method based only on these aspects. Thus, they inspected the code with *Source code view* to confirm that the function/methods provide certain functionality.

To detect the allocation hotspots, participants discovered the memory usage of functions/methods and compared the visual cues of nodes in *Call graph view* and *Sub call graph view*. However, participants sometimes struggle to identify allocation hotspots in *Call graph view* due to the presence of several nodes. For this reason, participants usually employed *Scatter plot view* to quickly found the allocation hotspots or confirm the expected allocation hotspots located previously. All participants also determined the code lines that allocated the most memory with *Source code view* by comparing the highlighted lines.

Moreover, when inspecting the circumstances, rationale, and events of selected functions/methods, participants used three

views. Participants employed *Call graph view* and *Sub call graph view* to explore the callers, callees, and the execution path of a particular function/method. Some participants mentioned that *Sub call graph view* was more suitable for detecting the situations involved in the execution of a function/method and navigating quickly and iteratively through the calling relationships compared to the *Call graph view*.

To comprehend the rationale and identify the memory events (allocations, accesses, releases) related to some functions/methods, participants explored the *Source code view*. Therefore, participants quickly discovered memory allocations following the highlighted lines. They also profoundly examined the code to identify and understand the memory accesses and releases since Vismep does not support these activities.

Understanding memory usage. All participants understood the memory consumed by their applications. They inspected the memory used of allocation hotspots and relevant code by hovering the cursor over the respective nodes in *Call graph view* and *Scatter plot view*. Next, they verified if the memory consumed by those functions/methods was the expected. Consequently, some participants detected (i) unexpected allocation hotspots, (ii) relevant code that consume more or less memory than anticipated, and (iii) that most allocation hotspots are involved with external libraries (*e.g.,* pandas, numpy).

When looking for memory anomalies, participants first located the allocation hotspots and relevant code in the *Call graph view* and *Scatter plot view*. Next, participants analyzed the memory used by those functions/methods to determine if the memory consumed was excessive or unnecessary, considering the correct program functionality. Four participants did not detect any memory anomaly since most allocation hotspots belong to external libraries, and the memory used was considered not excessive. The remaining participants checked the circumstances, rationale, and events of the allocation hotspots and relevant code to locate any unnecessary and unexcepted memory behavior. Thus, participants analyzed the number of executions, memory usage, and under what circumstances those functions/methods are executed using the *Call graph view*. Additionally, some participants examined the rationale and memory events in the *Source code view* to estimate if the memory usage is reasonable and if the memory allocations are essential.

Optimizing memory usage. Participants reduced or tried to reduce memory usage by analyzing the circumstances, rationale, and events of memory anomalies or allocation hotspots. Participants employed *Call graph view* to inspect the circumstances in which these functions/methods are executed. Besides, some participants explored the *Source code view* to analyze the memory events and how changing some code lines could affect memory usage and functionality of relevant code.

Furthermore, four participants did not locate any memory anomaly and could not find any optimization opportunity. These participants mentioned that to reduce the memory consumption, they required more knowledge and time to fully comprehend the circumstances, rationale, and events about allocation hotspots that belong to external libraries.

Seven participants identified anomalies cause. As a result, they found (i) the use of unsuitable data structure, (ii) unnecessary memory allocations, and (iii) the presence of temporary allocations (allocations created and released from memory several times). We must mention that participants did not modify their code using Vismep since this activity is not supported yet. As a result, they changed the source code program using an IDE or a text editor. However, only four successfully modified the code with the information from anomalies cause. These participants used Vismep again to inspect the memory usage of the allocation hotspots or relevant code using *Call graph view* or *Scatter plot view*. The remaining participants had problems programming the optimizations since these changes negatively impacted the application's functionality.

RQ1.2: Overall, programmers used various views to obtain information about memory usage. To detect relevant code and allocation hotspots, participants explored the *Call graph view*. Besides, they used the *Scatter plot view* to confirm the allocation hotspots found in other views. To inspect the circumstances, rationale, and events of particular functions/methods, most participants used the *Call graph view*, but some of them indicated that *Sub call graph view* was more suitable for this activity. To locate memory anomalies, participants looked for the information previously mentioned via the *Call graph view* and *Source code view*. Finally, participants detected anomalies cause by inspecting the calling relationships in *Call graph view* and analyzing how anomalies affect the functionality with the *Source code view*.

C. RQ2.1: Cognitive load

Table III shows ranges and mean values of the overall and dimensions TLX scores over Vismep. NASA-TLX score ranges from 0 (low mental workload) to 100 (high mental workload). The average task load index reported by participants using Vismep for memory consumption analysis is 29.69 (std. dev. 14.07). According to Grier [32] and Hertzum [33], this indicates a low to moderate effort.

	Min	Max	Mean	SD
Overall TLX	11.66	56.66	29.69	14.07
Dimensions				
Mental demand	10	80	44.54	26.21
Physical demand	0	70	14.54	20.67
Temporal demand	10	70	42.72	17.93
Performance	0	50	16.36	15.66
Effort	20	80	43.63	23.77
Frustration	0	40	16.36	16.89

TABLE III: Ranges and means of overall workload and dimensions TLX scores.

The score for dimensions varies from 0 (low demand) to 100 (high demand), except for the performance, which ranges from 0 (high overall performance) to 100 (low overall performance). We identified that mental demand, temporal demand, and effort means are the highest among all dimensions. We consider that these scores reflect the issues that some participants mentioned when locating allocation hotspots and inspecting the circumstances, rationale, and events in a *Call graph view* with several functions/methods (mostly from external libraries). It also points out that although Vismep indicates useful information and satisfies some needs, one factor that negatively impacts practitioners is Vismep's performance mentioned in Section V-E.

RQ2.1: Participants often perceive a low to moderate mental workload effort using Vismep. Besides, the mental demand, temporal demand, and effort could be reduced by improving the profiler and Vismep support for specific activities.

D. RQ2.2: Perception of usability

Table IV illustrates ranges and mean values of the SUS score and components of SUS scores associated with Vismep. SUS score varies from 0 (worst imaginable) to 100 (excellent). The average SUS score calculated from the participant's answers is 72.5 (std. dev. 7.98). According to Sauro [34] Vismep is graded "C+" which indicates a "good" usability score.

	Min	Max	Mean	SD
Overall SUS	60	82.5	72.5	7.98
Usability aspects				
Q1: Willing to use the tool	3	5	3.91	0.53
Q2: Complexity of the tool	1	3	1.90	0.53
Q3: Ease of use	3	5	4	0.45
Q4: Need of support to use	2	5	3.09	1.13
Q5: Integrity of functions	3	5	4	0.45
Q6: Inconsistency	1	3	1.91	0.83
Q7: Intuitiveness	2	5	3.82	0.98
Q8: Cumbersomeness to use	1	4	1.63	0.92
Q9: Feeling confident to use	2	5	3.73	0.90
Q10: Required learning-effort	1	3	1.90	0.70

TABLE IV: Ranges and means of overall SUS and components of SUS scores.

We detailed the scores for the components of SUS to understand the participant's perception of the different aspects of usability. The score for components ranges from 1 to 5. These components represent positive aspects (*i.e.,* Q1, Q3, Q5, Q7, and Q9) and negative aspects (*i.e.,* Q2, Q4, Q6, Q8, and Q10) of usability. We detected that Vismep achieved higher scores in positive aspects and lower scores for negative aspects (except for the need of support to use Vismep). Section V-E details that most programmers need support to use Vismep because they were unsure about (i) how to run the profiler and (ii) the state of the profiler (*e.g.,* still running or stopped for an issue).

RQ2.2: Participants perceived that Vismep provides high overall usability considered *"good"*. Improving the profiler and the learning material could reduce the requirement of support when a user employs Vismep.

E. RQ2.3: Perception of Vismep features

We identified nine general themes by using grounded theory [31] to process the feedback. Each theme is presented with its name and the format $[O/P]$, where O indicates the number of times a theme occurs in the sessions and P denotes the number of participants who explicitly expressed it. First, we consider the following themes as positive:

- *Useful views and interactions.* [24/11] Participants mentioned that Vismep provides useful views and interactions to analyze the memory consumed by their applications. They described that some views were suitable for quickly locating relevant code (*Call graph view*), allocation hotspots (*Scatter plot view*), among others. Additionally, the views assist programmers in comprehending diverse aspects (*e.g.,* memory usage, number of executions, calls between functions/methods), as well as locating memory anomalies or unexpected behaviors. Participants also navigated over various views iteratively was helpful to inspect circumstances, rationale, and events of relevant code and allocation hotspots quickly and trace anomalies cause.
- *Visual aspects.* [11/8] Participants highlighted some positive points over the visual cues. They indicated that visualizations were intuitive and easy to use due to the visual mapping. For instance, some programmers mentioned that the *Call graph view* offers a good overview of the application. Additionally, they emphasized that the *Source code view* and *Sub call graph view* provided visual cues that support the inspection of circumstances, rationale, and events of selected functions/methods.
- *Connection with source code.* [10/7] Participants indicated that connecting the dynamic aspects with the source code helped them comprehend program behavior, memory events associated with a function/method, and discover memory anomalies. Participants highlighted the facilities over navigating between the *Source code view* and other views.
- *Usability.* [6/5] Participants said that Vismep was easy to use, intuitive, and useful for analyzing memory usage. As a result, some participants indicated that they would like to use the prototype daily.

We also detected themes that we considered negative points of Vismep:

- *Opportunities for improvement.* [27/10] Most participants made suggestions on various aspects of Vismep. To illustrate, eight participants indicated that filtering functions/methods based on criteria (*e.g.,* module, memory usage) would facilitate the navigation in *Call graph view* and locating allocation hotspots and relevant code. On the other

hand, three participants said that adding a message to show the profiler progress and improving the tutorial would reduce the need for support in using the prototype. Besides, they commented that improving Vismep's performance would help reduce temporal demand.
- *Missing information.* [10/6] Participants indicated that it would be helpful to provide information regarding (i) distribution of memory over the function/methods, (ii) memory evolution over time, and (iii) allocations made over time and their memory usage.
- *Bugs.* [6/4] Participants also detected some bugs. For example, three participants identified issues in Vismep when the source code of functions/methods from external libraries were displayed. Also, two participants mentioned problems with some interactions (*e.g.,* static highlighted nodes).
- *Metric selection.* [4/3] Participants suggested that letting users modify the visual mapping, scales, and displayed metrics could be very useful for quickly locating the information required.
- *Integration with IDE.* [2/2] Participants suggested that it would be useful to present Vismep features integrated with a programming environment.

RQ2.3: Overall, participants appreciated the views and usability offered by Vismep. We also located some opportunities to improve Vismep, such as (i) adding new interactions, (ii) considering new information (memory usage over time), and (ii) fixing bugs.

VI. Threats to validity

We identified and organized some threats to our research's validity based on the work of Wohlin *et al.* [35].

Conclusion Validity. The individual differences among participants, the sample size and the use of Vismep could impact our conclusion. Therefore, our conclusion might not be representative. Consequently, our results could be different given other tools or participants. We try to reduce this threat by selecting programmers with different backgrounds and experience levels. However, an additional study that involves more people and other tools may mitigate this threat.

Internal Validity. Participants were not familiar with Vismep prior to the study. The latter may restrict participants' effective use of Vismep for memory usage analysis, causing a low SUS score and a high mental workload. The exploration period was part of the study to mitigate this threat. However, the score for question 4 in SUS form (Table IV) indicates a need for help when using Vismep.

Regarding data collection and analysis for RQ1, an author generated the spreadsheets and identified the information needs. Another author checked if the spreadsheets and the information needs detected were consistent with the audio, video records, and tracking logs to minimize biases during the process.

Construct Validity. We voluntarily centered on the Python programming language. Participants selected applications under analysis with which they were familiar. Data from each work session was carefully examined and collected using records, logs and observation while participants tackled a particular question.

VII. Conclusion and future work

This study introduces Vismep, an interactive visualization prototype that helps programmers analyze the memory usage of Python applications, and presents an exploratory study involving eleven participants that used Vismep to analyze the memory usage of their projects. Our results show that programmers need to explore dynamic and static information to (a) locate relevant code, (b) identify allocation hotspots, (c) inspect the circumstances, rationale, and events of functions/methods, (d) infer memory anomalies, and (e) trace the cause of anomalies.

We also noticed that participants used different Vismep views or combined some of them to perform previously mentioned activities. Some participants explained that some views are more suitable for some activities, such as *Scatter plot view* to identify functions that allocate most memory (allocation hotspots) or *Sub call graph view* to explore the control flow (circumstances). Additionally, we reported when participants struggled using Vismep to perform the activities. We detected missing information that users required and possibilities to improve the design of Vismep and other tools. Finally, we noted that Vismep is positively perceived because participants indicated a low to moderate mental workload effort when using it and estimated that Vismep offers high overall usability.

Limitations and future work. Since Vismep profiler is based on two popular modules, the profiler could present some issues involving these modules (*i.e.,* execution time, accuracy of memory usage). Thus, we plan to modify our profiler to decrease the execution time and extract more detailed data about memory events. We also plan to improve the design of the visualizations based on the information collected at Section V-B and Section V-E.

Furthermore, we presented this exploratory study to show how programmers use Vismep to inspect applications. As future work, we plan to conduct a study to compare the performance of participants who use Vismep with the performance of those who employ other tools.

Acknowledge

Alison Fernandez Blanco is supported by a Ph.D. scholarship from CONICYT, Chile. CONICYT-PFCHA/Doctorado Nacional/2019-21191851. Alexandre Bergel is grateful to the ANID FONDECYT Regular project 1200067 for having partially sponsored the work presented in this article. Juan Pablo Sandoval Alcocer thanks ANID FONDECYT Iniciación Folio 11220885 for supporting this article.

References

[1] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, C. Zhai, Bug characteristics in open source software, Empirical Software Engineering 19 (6) (2014) 1665–1705. doi:10.1007/s10664-013-9258-8.
URL https://doi.org/10.1007/s10664-013-9258-8

[2] M. Ghanavati, D. Costa, J. Seboek, D. Lo, A. Andrzejak, Memory and resource leak defects and their repairs in java projects, Empirical Software Engineering 25 (1) (2020) 678–718. doi:10.1007/s10664-019-09731-8.
URL https://doi.org/10.1007/s10664-019-09731-8

[3] N. Mitchell, G. Sevitsky, The causes of bloat, the limits of health, SIGPLAN Not. 42 (10) (2007) 245–260. doi:10.1145/1297105.1297046.
URL https://doi.org/10.1145/1297105.1297046

[4] M. Weninger, E. Gander, H. Mössenböck, Analyzing data structure growth over time to facilitate memory leak detection, in: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE '19, Association for Computing Machinery, New York, NY, USA, 2019, p. 273–284. doi:10.1145/3297663.3310297.
URL https://doi.org/10.1145/3297663.3310297

[5] N. Nethercote, J. Seward, Valgrind: A framework for heavyweight dynamic binary instrumentation, SIGPLAN Not. 42 (6) (2007) 89–100. doi:10.1145/1273442.1250746.
URL https://doi.org/10.1145/1273442.1250746

[6] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, K. Hazelwood, Pin: Building customized program analysis tools with dynamic instrumentation, SIGPLAN Not. 40 (6) (2005) 190–200. doi:10.1145/1064978.1065034.
URL https://doi.org/10.1145/1064978.1065034

[7] M. Weninger, L. Makor, H. Mossenbock, Memory cities: Visualizing heap memory evolution using the software city metaphor, in: 2020 Working Conference on Software Visualization (VISSOFT), IEEE Computer Society, Los Alamitos, CA, USA, 2020, pp. 110–121. doi:10.1109/VISSOFT51673.2020.00017.
URL https://doi.ieeecomputersociety.org/10.1109/VISSOFT51673.2020.00017

[8] S. Byma, J. R. Larus, Detailed heap profiling, SIGPLAN Not. 53 (5) (2018) 1–13. doi:10.1145/3299706.3210564.
URL https://doi.org/10.1145/3299706.3210564

[9] A. Fernandez Blanco, J. P. S. Alcocer, A. Bergel, Effective visualization of object allocation sites, in: 2018 IEEE Working Conference on Software Visualization (VISSOFT), 2018, pp. 43–53. doi:10.1109/VISSOFT.2018.00013.

[10] P. Gralka, C. Schulz, G. Reina, D. Weiskopf, T. Ertl, Visual exploration of memory traces and call stacks, in: 2017 IEEE Working Conference on Software Visualization (VISSOFT), 2017, pp. 54–63. doi:10.1109/VISSOFT.2017.15.

[11] M. Lanza, S. Ducasse, Polymetric views - a lightweight visual approach to reverse engineering, IEEE Transactions on Software Engineering 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.

[12] S. Ducasse, M. Lanza, R. Bertuli, High-level polymetric views of condensed run-time information, in: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., 2004, pp. 309–318. doi:10.1109/CSMR.2004.1281433.

[13] G. Xu, A. Rountev, Precise memory leak detection for java software using container profiling, ACM Trans. Softw. Eng. Methodol. 22 (3) (jul 2013). doi:10.1145/2491509.2491511.
URL https://doi.org/10.1145/2491509.2491511

[14] D. Lo, N. Nagappan, T. Zimmermann, How practitioners perceive the relevance of software engineering research, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Association for Computing Machinery, New York, NY, USA, 2015, p. 415–425. doi:10.1145/2786805.2786809.
URL https://doi.org/10.1145/2786805.2786809

[15] M. Lanza, S. Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795. doi:10.1109/TSE.2003.1232284.
URL http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf

[16] M. Weninger, P. Grünbacher, E. Gander, A. Schörgenhumer, Evaluating an interactive memory analysis tool: Findings from a cognitive walkthrough and a user study, Proc. ACM Hum.-Comput. Interact. 4 (EICS) (jun 2020). doi:10.1145/3394977.
URL https://doi.org/10.1145/3394977

[17] F. Duseau, B. Dufour, H. Sahraoui, Vasco: A visual approach to explore object churn in framework-intensive applications, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 15–24. doi:10.1109/ICSM.2012.6405248.

[18] S. Moreta, A. Telea, Visualizing dynamic memory allocations, in: 2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis, 2007, pp. 31–38. doi:10.1109/VISSOF.2007.4290697.

[19] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, J. Yang, Visualizing the execution of java programs, in: S. Diehl (Ed.), Software Visualization, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 151–162.

[20] W. De Pauw, G. Sevitsky, Visualizing reference patterns for solving memory leaks in java, in: R. Guerraoui (Ed.), ECOOP' 99 — Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 116–134.

[21] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, S. Z. Guyer, Heapviz: Interactive heap visualization for program understanding and debugging, in: Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 53–62. doi:10.1145/1879211.1879222.
URL https://doi.org/10.1145/1879211.1879222

[22] A. F. Blanco, A. Bergel, J. P. S. Alcocer, Software visualizations to analyze memory consumption: A literature review, ACM Comput. Surv. 55 (1) (jan 2022). doi:10.1145/3485134.
URL https://doi.org/10.1145/3485134

[23] J. P. Sandoval Alcocer, H. Camacho Jaimes, D. Costa, A. Bergel, F. Beck, Enhancing commit graphs with visual runtime clues, in: 2019 Working Conference on Software Visualization (VISSOFT), 2019, pp. 28–32. doi:10.1109/VISSOFT.2019.00012.

[24] A. N. M. I. Choudhury, P. Rosen, Abstract visualization of runtime memory behavior, in: 2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT), 2011, pp. 1–8. doi:10.1109/VISSOF.2011.6069452.

[25] R. Ishizue, K. Sakamoto, H. Washizaki, Y. Fukazawa, Pvc.js: visualizing c programs on web browsers for novices, Heliyon 6 (4) (2020) e03806. doi:https://doi.org/10.1016/j.heliyon.2020.e03806.
URL https://www.sciencedirect.com/science/article/pii/S2405844020306514

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

[27] S. G. Hart, L. E. Staveland, Development of nasa-tlx (task load index): Results of empirical and theoretical research, in: Advances in psychology, Vol. 52, Elsevier, 1988, pp. 139–183.

[28] A. Bangor, P. T. Kortum, J. T. Miller, An empirical evaluation of the system usability scale, International Journal of Human–Computer Interaction 24 (6) (2008) 574–594. doi:10.1080/10447310802205776.
URL https://doi.org/10.1080/10447310802205776

[29] J. Saldana, The Coding Manual for Qualitative Researchers, Core textbook, SAGE Publications, 2021.
URL https://books.google.cl/books?id=RwcVEAAAQBAJ

[30] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, C. Kästner, On debugging the performance of configurable software systems: Developer needs and tailored tool support, arXiv preprint arXiv:2203.10356 (2022).

[31] J. M. Corbin, A. C. Strauss, Basics of qualitative research, 3rd Edition, SAGE Publications, Thousand Oaks, CA, 2008.

[32] R. A. Grier, How high is high? a meta-analysis of nasa-tlx global workload scores, Proceedings of the Human Factors and Ergonomics Society Annual Meeting 59 (1) (2015) 1727–1731. doi:10.1177/1541931215591373.
URL https://doi.org/10.1177/1541931215591373

[33] M. Hertzum, Reference values and subscale patterns for the task load index (tlx): a meta-analytic review, Ergonomics 64 (7) (2021) 869–878. doi:10.1080/00140139.2021.1876927.
URL https://doi.org/10.1080/00140139.2021.1876927

[34] J. Sauro, A practical guide to the system usability scale: Background, benchmarks & best practices, Measuring Usability LLC, 2011.

[35] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering, Springer Science & Business Media, 2012.