



UNIVERSIDAD DE CHILE  
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS  
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

EMPIRICAL FOUNDATION FOR MEMORY USAGE ANALYSIS THROUGH  
SOFTWARE VISUALIZATIONS

TESIS PARA OPTAR AL GRADO DE  
DOCTORA EN COMPUTACIÓN

ALISON FERNANDEZ BLANCO

PROFESOR GUÍA:  
ALEXANDRE BERGEL

PROFESOR CO-GUÍA:  
JUAN PABLO SANDOVAL ALCOCER

MIEMBROS DE LA COMISIÓN:  
JOCELYN SIMMONDS WAGEMANN  
LUIS MATEU BRULE  
HOUARI SAHRAOUI

Este trabajo ha sido parcialmente financiado por CONICYT-PFCHA/Doctorado  
Nacional/2019-21191851.

SANTIAGO DE CHILE  
2023

RESUMEN DE LA TESIS PARA OPTAR  
AL GRADO DE DOCTORA EN COMPUTACION  
POR: ALISON FERNANDEZ BLANCO  
FECHA: 2023  
PROF. GUIA: ALEXANDRE BERGEL  
PROF. CO-GUIA: JUAN PABLO SANDOVAL ALCOCER

## **FUNDAMENTO EMPÍRICO PARA EL ANÁLISIS DEL USO DE LA MEMORIA A TRAVÉS DE VISUALIZACIONES DE SOFTWARE**

Los desarrolladores a menudo pasan mucho tiempo monitoreando manualmente el uso de memoria para localizar anomalías (p. ej., fugas, sobrecargas de memoria) que puedan generar fallas en las aplicaciones de software. Por esta razón, se han propuesto herramientas que proporcionan una amplia gama de información a través de informes de texto o visualizaciones. Sin embargo, todavía hay poca comprensión de las necesidades del programador al analizar el uso de la memoria, qué tan bien las herramientas y los enfoques actuales ayudan a los usuarios en este proceso y la percepción que los programadores tienen de las herramientas.

En esta tesis, llevamos a cabo una revisión sistemática sobre visualizaciones de software enfocadas en el análisis del uso de la memoria con el objetivo de organizar e introducir una taxonomía basada en cinco dimensiones relevantes. Como resultado, este estudio destaca (i) las principales características de los enfoques visuales actuales, (ii) los desafíos del campo y (iii) una serie de áreas de investigación que vale la pena explorar. Con base en este estudio, proponemos Vismep, un prototipo de visualización interactiva para ayudar a los programadores a analizar el uso de memoria de las aplicaciones de Python. Basamos el diseño de Vismep en las características comunes utilizadas en las áreas más modernas y en algunos aspectos que valen la pena explorar. También presentamos un estudio exploratorio para comprender cómo los programadores emplean Vismep para analizar el uso de memoria de las aplicaciones de Python y su percepción de Vismep. Nuestros hallazgos ilustran que los programadores usan información dinámica y estática para satisfacer cinco necesidades. Además, reportamos el uso de Vismep para la obtención de la información requerida, los desafíos enfrentados durante el proceso y la percepción de esfuerzo de carga mental y usabilidad. Para entender con mayor precisión las necesidades de los programadores a la hora de analizar el consumo de memoria, realizamos un estudio más exhaustivo utilizando Vismep y Tracemalloc, el perfilador de memoria estándar en Python. Como resultado, proporcionamos un catálogo de 34 preguntas que los programadores se hacen al analizar el consumo de memoria. También presentamos un análisis detallado del uso de Vismep y Tracemalloc para responder a estas preguntas y las dificultades que enfrentan los profesionales durante el proceso.

ABSTRACT OF THE THESIS TO OBTAIN  
THE GRADE OF DOCTOR IN COMPUTER  
BY: ALISON FERNANDEZ BLANCO  
DATE: 2023  
ADVISOR: ALEXANDRE BERGEL  
CO-ADVISOR: JUAN PABLO SANDOVAL ALCOCER

## **EMPIRICAL FOUNDATION FOR MEMORY USAGE ANALYSIS THROUGH SOFTWARE VISUALIZATIONS**

Developers often spend substantial time manually monitoring memory consumption to localize memory anomalies (*e.g.*, memory leaks, memory bloats) that usually generate crashes on software applications. For this reason, a number of memory profiling tools have been proposed providing a wide range of information displayed through full-text reports or visualizations. However, there is still little understanding of the programmer's needs when analyzing memory usage, how well current tools and approaches support users in this process and the practitioners' perception of the tools.

In this thesis, we conducted a systematic literature review about software visualizations for memory usage analysis to organize and introduce a taxonomy based on five dimensions relevant to software visualizations. As a result, this study highlights (i) the main characteristics of current visual approaches, (ii) the challenges of the field, and (iii) a number of research areas that are worth exploring. Based on this study, we propose Vismep, an interactive visualization prototype to help programmers analyze Python applications' memory usage. We based Vismep's design on the common characteristics used on state-of-the-art and some areas worthy of exploring. We also present an exploratory study to understand how programmers employ Vismep to analyze the memory usage of Python applications and their perception of Vismep. Our findings illustrate that programmers use dynamic and static information to satisfy five needs. In addition, we reported the Vismep usage for obtaining the required information, the challenges faced during the process, and the perception of mental workload effort and usability. In order to understand more precisely the programmers' needs when analyzing memory consumption, we conducted a more exhaustive study using Vismep and Tracemalloc, the standard memory profiler in Python. As a result, we provide a catalog of 34 questions programmers ask themselves when analyzing memory consumption. We also present a detailed analysis of the use of Vismep and Tracemalloc to answer these questions and the difficulties that practitioners face during the process.

*A mi familia*  
*To my family*

# Acknowledgments

Quiero agradecer a Dios por darme la oportunidad y el tiempo para crecer en el ámbito personal así como el profesional y para concluir esta etapa de mi vida.

Gracias a mis padres Ivan y Leticia, a mi hermana Adriana y al resto de mi familia que sin lugar a dudas me apoyaron durante el traslado a Chile y durante el desarrollo de la tesis. Muchas gracias por sus palabras de aliento y su buena voluntad.

Agradecer a mis tutores, Alexandre Bergel y Juan Pablo Sandoval Alcocer, por brindarme su apoyo y guía durante todo este proceso, sobre todo por confiar en mi. También a los miembros de la comisión: Jocelyn Simmonds, Luis Mateu y Houari Sahraoui; les agradezco por sus comentarios que fueron de mucha ayuda para mejorar mi trabajo.

Gracias a Javier Ojeda por tu apoyo incondicional, tus sugerencias y tu retroalimentación durante estos años. Muchas gracias a mis amigos por su compañía durante estos años, en particular por su alegría, su apoyo y su honestidad. Agradezco al equipo ISCLab, son muchos para nombrarlos a todos, pero gracias por apoyarme y darme consejos para mi trabajo. Agradezco la comprensión y la guía de Angelica Aguirre y Sandra Gaez que amorosamente ayudan a los estudiantes. Gracias a Renato Cerro por toda la ayuda en el area de inglés. Y por supuesto, la lista no estaría completa sin mencionar a toda la gente del Departamento de Ciencias de la Computación que ocupa diferentes puestos y tiene distintos deberes, manteniendo un agradable ambiente para la educación y la investigación.

Quiero agradecer a CONICYT, por financiar mis estudios de doctorado a través de la beca CONICYT-PFCHA/Doctorado Nacional/2019-21191851. También agradecer a NIC Chile por el apoyo brindado durante mi primer semestre.

# Table of Content

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Analyzing Memory Usage . . . . .	3
1.2	Visualizing Memory Consumption . . . . .	4
1.3	Understanding the Problem of Visualizing Memory Usage . . . . .	5
1.4	Our Proposal . . . . .	6
1.4.1	Research goals . . . . .	6
1.5	Contributions . . . . .	6
1.6	Vismep Overview . . . . .	7
1.7	Scope and Limitations . . . . .	8
1.8	Related Publications . . . . .	9
1.9	Thesis Outline . . . . .	10
<b>2</b>	<b>Software Visualizations to Analyze Memory Consumption</b>	<b>11</b>
2.1	Introduction . . . . .	11
2.2	Methodology . . . . .	12
2.2.1	Research Questions . . . . .	13
2.2.2	Search Strategy . . . . .	14
2.2.3	Inclusion & Exclusion Criteria . . . . .	16
2.2.4	Quality Assessment . . . . .	17
2.2.5	Data Extraction . . . . .	18
2.2.6	Data Analysis . . . . .	19

2.3	Results . . . . .	21
2.3.1	RQ1: Problems Domain . . . . .	23
2.3.2	RQ2: Data . . . . .	28
2.3.3	RQ3: Visual Representation . . . . .	31
2.3.4	RQ4: Evaluation . . . . .	38
2.3.5	RQ5: Availability . . . . .	39
2.4	Discussion and Open Challenges . . . . .	41
2.5	Related Work . . . . .	43
2.6	Threats to validity . . . . .	45
2.7	Summary . . . . .	46
<b>3</b>	<b>Visualizing Memory Consumption with Vismep</b>	<b>48</b>
3.1	Introduction . . . . .	48
3.2	Related Work . . . . .	50
3.2.1	Memory Consumption Analysis in Python . . . . .	50
3.2.2	Studies on Software Visualization for Memory Usage Analysis . . . . .	52
3.2.3	Software Visualization Evaluation . . . . .	52
3.3	Vismep . . . . .	53
3.3.1	Overview . . . . .	53
3.3.2	Vismep In a Nutshell: Exploring a Pandas Issue . . . . .	55
3.3.3	Call graph view . . . . .	55
3.3.4	Source Code View . . . . .	57
3.3.5	Sub Call Graph View . . . . .	58
3.3.6	Scatter Plot View . . . . .	58
3.3.7	Interactions . . . . .	59
3.4	Methodology . . . . .	60
3.4.1	Research Questions . . . . .	61
3.4.2	Participants & Applications . . . . .	61

3.4.3	Procedure . . . . .	63
3.4.4	Data Collection and Transcription . . . . .	65
3.4.5	Data Analysis . . . . .	66
3.5	Results . . . . .	66
3.5.1	RQ1.1: Information needs . . . . .	67
3.5.2	RQ1.2: Use of Vismep . . . . .	68
3.5.3	RQ2.1: Cognitive Load . . . . .	72
3.5.4	RQ2.2: Perception of usability . . . . .	72
3.5.5	RQ2.3: Perception of Vismep features . . . . .	73
3.6	Discussion . . . . .	75
3.7	Threats to validity . . . . .	76
3.8	Summary . . . . .	77
<b>4</b>	<b>Answering and Asking Questions During Memory Consumption Analysis</b>	<b>79</b>
4.1	Introduction . . . . .	80
4.2	Related Work . . . . .	81
4.3	Methodology . . . . .	82
4.3.1	Research Questions . . . . .	82
4.3.2	Memory Profiler Tools . . . . .	83
4.3.3	Participants & Applications . . . . .	86
4.3.4	Procedure . . . . .	88
4.3.5	Data Collection and Transcription . . . . .	90
4.3.6	Data Analysis . . . . .	90
4.4	Results: What Questions do Python Programmers Ask During Memory Usage Analysis? . . . . .	92
4.4.1	Understanding Static Structure, Intent and Implementation . . . . .	93
4.4.2	Understanding Control Flow . . . . .	95
4.4.3	Discovering Memory Usage in a Single Point of Time . . . . .	95



4.4.4	Comparing and Contrasting Memory Usage . . . . .	96
4.4.5	Discovering Memory Events . . . . .	97
4.5	Results: How do Python Programmers Answer These Questions Using Vismep and Tracemalloc? . . . . .	98
4.5.1	Understanding Static Structure and Implementation . . . . .	98
4.5.2	Understanding Control Flow . . . . .	100
4.5.3	Discovering Memory Usage at a Single Point of Time . . . . .	101
4.5.4	Comparing and Contrasting Memory Usage . . . . .	102
4.5.5	Discovering Memory Events . . . . .	104
4.5.6	Unanswered questions . . . . .	105
4.5.7	Suggestions . . . . .	107
4.6	Discussion . . . . .	108
4.6.1	Questions asked by participants . . . . .	108
4.6.2	Answering questions . . . . .	110
4.7	Threats to validity . . . . .	112
4.8	Summary . . . . .	113
<b>5</b>	<b>Conclusions</b>	<b>115</b>
5.1	Dissertation Contributions . . . . .	115
5.2	Limitations . . . . .	118
5.3	Empirical Foundation Impact . . . . .	122
5.4	Future Work . . . . .	123
	<b>Bibliography</b>	<b>139</b>
	<b>Annexes</b>	<b>140</b>
<b>A</b>	<b>Search String for Digital Libraries</b>	<b>141</b>
<b>B</b>	<b>Perception of Vismep and Tracemalloc</b>	<b>142</b>

B.1 Cognitive Load . . . . .	142
B.2 Usability . . . . .	143

# List of Tables

2.1	Research Questions . . . . .	13
2.2	An overview to the relations between our dimensions and the dimensions proposed by some works of the state-of-art. . . . .	14
2.3	Search terms and alternatives of spelling . . . . .	15
2.4	Inclusion and Exclusion Criteria . . . . .	17
2.5	Quality assessment adopted from [147] . . . . .	18
2.6	Classification scheme . . . . .	20
2.7	Systematic Search Results . . . . .	21
2.8	The included papers in the study (S1-S30) . . . . .	22
2.9	The included papers in the study (S31-S46) . . . . .	23
2.10	Classification of articles based on the tasks . . . . .	24
2.11	Classification of articles based on the data source . . . . .	28
2.12	Classification of articles based on the visual technique . . . . .	32
2.13	Classification of articles based on the strategies used to evaluate visualizations . . . . .	38
2.14	Visualization tools and additional information from the selected articles. The information was verified on 18/05/2021. . . . .	40
3.1	Libraries and tools along with (i) the activities that claim to support (A1 = Analyzing memory usage of entities; A2 = Analyzing allocation hotspots; A3 = Analyzing memory usage over time; A4 = Analyzing leaking objects), (ii) the information reported (M.A. = Memory allocations; M.R = Memory releases; R.F. = Relationships between functions; V.R. = Variable references; T. = Time; TH. = Threads; L.C. = Lines of code; C. = Class; S.C. = Structural component) and (iii) the report presentation used. The information was verified at 02/09/2022. . . . .	51

3.2	Information of participants (Python programming experience (years); Self-assessment expertise (Likert-scale: 1 (novice) to 5 (expert)); Experience in memory usage analysis; Experience in addressing memory issues; Strategies when analyzing memory usage). . . . .	62
3.3	Questions answered by the participants. . . . .	63
3.4	Information needs, actions performed, and views (CGV = Call graph view; SCV = Source code view; SPV = Scatter plot view; SCGV = Sub call graph view) in Vismep that the eleven participants explored to analyze applications from Python. . . . .	67
3.5	Ranges and means of overall workload and dimensions TLX scores. . . . .	72
3.6	Ranges and means of overall SUS and components of SUS scores. . . . .	73
4.1	Information of participants (Python programming experience (years); Self-assessment expertise (Likert-scale: 1 (novice) to 5 (expert)); Experience in memory usage analysis; Experience in addressing memory issues; Activities when analyzing memory usage). Participants from groups G1 and G2 present gray and white backgrounds, respectively. . . . .	87
4.2	Questions made to the participant. . . . .	88
4.3	Questions per category and the occurrences raised during the work sessions. Each column corresponding to a participant presents the format: T (A/B), where A denotes the number of occurrences using Vismep, B indicates the number of occurrences using Tracemalloc, and T denotes the total number of occurrences. Note that if T is zero, the cell is empty. . . . .	94
A.1	Search query for the three digital libraries . . . . .	141
B.1	Means of overall workload and dimensions TLX scores for G1 and G2 using Vismep and Tracemalloc. . . . .	143
B.2	Means of overall SUS and components of SUS scores for G1 and G2 using Vismep and Tracemalloc. . . . .	143

# List of Figures

1.1	The memory issue #45489 reported in the pandas package that shows the code example which generates a memory leak due to keeping references to no longer needed objects. . . . .	2
1.2	The traceback of the object that allocates most memory of the running example of memory issue #45489 reported in the pandas package. The report shows the part of code and the execution path which allocate the most memory during program execution. . . . .	4
1.3	(a) A code example that illustrates a canvas with many shapes and performs an unnecessary computational operation to verify if the canvas is empty. (b) The main view of Vismep for running example of canvas, and (c) running example without the memory issue. . . . .	8
1.4	The main view of Vismep for running example in Figure 1.1 (right) and running example without the memory issue (left). . . . .	9
2.1	Overview of the workflow of the literature review. . . . .	13
2.2	The 46 included papers by publication year. . . . .	24
2.3	Visualization proposed by De Pauw <i>et al.</i> [39] to illustrate the references between the objects not reclaimed by the garbage collector in Java. ©2002 “Visualizing the Execution of Java Programs” by Wim De Pauw <i>et al.</i> . Reproduced with permission from Springer Nature. . . . .	26
2.4	AntTracks [156] provides overview plots about the application’s memory footprint and garbage collector overhead and allows practitioners to explore a time window. ©Used with permission of ACM (Association for Computing Machinery), from “Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study” by Markus Weninger <i>et al.</i> , 2020; permission conveyed through Copyright Clearance Center, Inc. . . . .	27

2.5	An example of <i>Memory cities</i> [157] representing the program’s heap shortly after startup (left) and 2 minutes and 300 garbage collections later (right). ©2020 Year IEEE. Reprinted, with permission, from “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor” by Markus Weninger <i>et al.</i> , 2020. . . . .	29
2.6	JProfiler displaying the methods executed with a Calling Context Tree. . . .	30
2.7	Gralka <i>et al.</i> [54] illustrate the application’s call stack over time as a flame graph. Each box corresponds to a method called in the application. In this case, colors are assigned based on modules. ©2017 Year IEEE. Reprinted, with permission, from “Visual Exploration of Memory Traces and Call Stacks” by Patrick Gralka <i>et al.</i> . . . . .	32
2.8	<i>Heapviz</i> [9] presents an interactive visualization to support the analysis of data structures. ©Republished with permission of ACM (Association for Computing Machinery), from “Heapviz: interactive heap visualization for program understanding and debugging”, by Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Permission conveyed through Copyright Clearance Center, Inc. . . . .	33
2.9	Visualization proposed by Sandoval [123] to analyze variations between commits. ©2019 Year IEEE. Reprinted, with permission, from “Enhancing Commit Graphs with Visual Runtime Clues” by Juan Pablo Sandoval Alcocer, 2019. . . . .	34
2.10	Visualization proposed by Moreta and Telea [93] to analyze memory allocations behavior. ©2007 Year IEEE. Reprinted, with permission, from “Visualizing Dynamic Memory Allocations” by Sergio Moreta, 2007. . . . .	35
2.11	<i>Vasco</i> [46], an interactive visualization to explore object churn. ©2012 Year IEEE. Reprinted, with permission, from “Vasco: A visual approach to explore object churn in framework-intensive applications” by Fleur Duseau, 2012. . . . .	36
3.1	The Vismep overview. The Vismep profiler collects the memory traces from the execution of a Python program and generates CSV files with the information. Our visualizer then reads the files generated and displays the main interactive view (Call graph view) through which the user can navigate and access the other complementary views. . . . .	53
3.2	Illustrating a memory issue of pandas with Vismep. <i>Call Graph view</i> (CG) shows the memory usage along the execution path. Each node denotes an executed function, and the edges indicate the calling relationships. When a function is selected, its border turns orange, and its source code is displayed in <i>Source Code view</i> (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. . . . .	56

3.3	On the right, the legend for <i>Call graph view</i> and <i>Sub call graph view</i> , the width of a node corresponds to the function’s average memory, and the height denotes the times a function is executed. On the left, an example with <i>Call graph view</i> , 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). . . . .	56
3.4	Overview of Sub call graph view. . . . .	58
3.5	Overview of Scatter plot view. . . . .	59
3.6	The user can navigate over different views of Vismep by continually interacting with each view. On the right, the user selects the <code>export_dataframe</code> function in <i>Call graph view</i> . Due to drill down, <code>export_dataframe</code> code is displayed in the middle view ( <i>Source code view</i> ). Then, the user selects the <code>to_json</code> called by <code>export_dataframe</code> to inspect the calling relationships of <code>to_json</code> function using <i>Sub call graph view</i> . . . . .	60
3.7	Overview of the workflow of the exploratory study. . . . .	61
3.8	During the pilot study, the first participant used the <i>Call graph view</i> (a) and the <i>Scatter plot view</i> (b) to identify the allocation hotspots. . . . .	64
3.9	During the pilot study, the sixth participant explores the node’s source code in the <i>Call graph view</i> . . . . .	65
3.10	The first participant focused on <code>main</code> function in the <i>Call graph view</i> . Then, she selected the respective node to understand the code of <code>main</code> function and detect lines of code responsible for allocating the most memory. . . . .	69
3.11	The fifth participant explored the calling relationships involved with <code>create_report</code> in <i>Sub call graph view</i> to determine under what circumstances this function is executed and if the memory allocated by this function is necessary. . . . .	70
4.1	Overview of the workflow of the exploratory study. . . . .	82
4.2	Enlisting the top ten memory allocation hotspots and total memory allocated using Tracemalloc. . . . .	83
4.3	Visualizing an example with Vismep (1) Call graph view – the main view that summarizes the functions with the calling relationships and the memory footprint,(2) Source code view – a view that displays the source code and highlights allocation hotspots, (3) Scatter plot view – a secondary view that shows the relationships between memory used and the execution times of functions, (4) Sub call graph view – for navigating through the calling relationships of a function. . . . .	84

4.4	P19 wanted to understand the rationale and implementation of <code>remove_tags_region</code> function. Consequently, the participant selected the respective node to display the view that contains the source code of <code>remove_tags_region</code> function and inspect the code. . . . .	99
4.5	P9 explores the calling relationships between functions in <i>Call graph view</i> and inspects which functions are called in the code using the <i>Source code view</i> . .	100
4.6	P8 used the API calls from <i>Display TOP</i> to enlist the parts of code that allocated the most memory and focused on the amount of memory allocated by specific lines of code. . . . .	102
4.7	P19 used the API calls from Compute differences to understand how the memory is allocated and released when a selected function is executed multiple times. The report shows the lines responsible for changing memory usage each time the function is executed. . . . .	103
4.8	P8 inspects the code of <code>SA</code> function using the <i>Source code view</i> to identify the memory allocations made in the function by focusing on the highlighted lines.	104



# Chapter 1

## Introduction


Memory space is a limited computational resource (*e.g.*, system on a chip, cloud computing), and nowadays, software applications manage a vast amount of data (*e.g.*, data science, machine learning, artificial intelligence) that increases over time. Software applications need to allocate memory to store data values and data structures. If a program allocates memory, never releases it, and runs for a sufficiently long time, it will eventually crash due to memory mismanagement. Therefore, developing tools or techniques that help programmers to analyze the memory consumption of programs and detect memory issues is part of the leading research ideas considered relevant in software engineering [76, 78, 90]. In the following, we enlist the factors that make memory consumption analysis and bug detection relevant and challenging:

***Memory management challenges.*** Regardless of whether memory management is manual or automatic, programmers may face memory issues. In some cases, like C and many other languages, programmers must manually manage memory allocation. Manual memory management is an error-prone process since developers may release memory too early or too late. For instance, a programmer attempting to use data values after they have been freed causes a fatal run-time error. Another common issue is not remembering to free allocated memory when it is no longer necessary; consequently, a long-running program will eventually crash since it will reach the limit on the available memory. On the other hand, Java, Ruby, and Python, among others, provide garbage collectors to manage memory automatically. The garbage collector tries to reclaim memory that was allocated by the program but is no longer referenced. Therefore, the garbage collectors relieve programmers from manual memory management and help programmers to avoid some issues. However, garbage collectors will not release the corresponding memory of unused objects that are still referenced [52]. Consequently, programmers may encounter memory issues even with the help of a garbage collector.


***The impact of memory issues.*** Memory issues are severe and usually make software applications crash and lead to performance degradation, such as in Mozilla, Apache, and the Linux kernel [52, 139]. Thus, memory issues should be detected and addressed at an early stage of development. However, memory bugs usually are not detected with the traditional testing processes and become perceptible in the production environment when the software application is more prone to manage a vast amount of data for a long time.

Furthermore, memory bugs are usually difficult to locate and repair since the programmer needs to (i) profoundly understand the program’s functionality, (ii) perform a careful examination of the source code, and (iii) collect and analyze a diverse set of dynamic and static aspects (*e.g.*, memory allocations, garbage collector information) at once [90, 165, 166]. Based on these points, some studies [78, 52, 139] indicate that programmers often spend considerable time and effort locating and fixing memory issues. These studies also reveal that memory bugs could be detected before the execution of a program by inspecting the source code manually. However, the most dangerous bugs are those observed when a programmer finds a performance degradation in a production environment, a raised memory error (out-of-memory), or a failing test. Ghanavati *et al.* [52] pinpoint that programmers verify the presence of memory bugs using memory profiler tools since the information these tools provide can considerably help developers reproduce and analyze the issue defects.

## Bug tojson memleak #45489

 Merged jreback merged 5 commits into `pandas-dev:main` from `vernetya:bug-tojson-memleak`

Conversation 7   Commits 5   Checks 39   Files changed 2

 **vernetya** commented on Jan 20 Contributor ...

- closes  **BUG: to\_json memory leak (introduced in 1.1.0) #43877**
- tests passed
- Ensure all linting tests pass, see [here](#) for how to run them
- whatsnew entry

Fix memory leak when calling `to_json` appeared in 1.1.0. From these [changes](#), in `get_values` function, it looks like `values` from [here](#) are not cleaned in this [path](#).

I run the following loop and plot the memory usage using [memory\\_profiler](#):

```
def foo():
    data = {str(c): list(range(100_000)) for c in range(10)}
    for i in range(500):
        print(i)
        df = pd.DataFrame(data)
        df.to_json(orient='split', indent=0)
```

Figure 1.1: The memory issue #45489 reported in the pandas package that shows the code example which generates a memory leak due to keeping references to no longer needed objects.

To illustrate more clearly the challenge of monitoring memory consumption and addressing memory issues, consider the memory issue reported in the pandas package<sup>1</sup>. Pandas is a flexible and powerful package for supporting programmers in data science/data analysis and

<sup>1</sup><https://github.com/pandas-dev/pandas/pull/45489>

machine learning tasks. The reported issue was reproducible with the code illustrated at Figure 1.1. The code essentially creates a dictionary (`data`), then a data frame object (`df`) is created based on `data` and converted into a JSON string several times. At first glance, it is a simple piece of code that allocates the necessary memory to fulfill its purpose. However, when GitHub users analyzed the code with memory profilers, they discovered that memory usage increases too much over time because it keeps holding references to objects that are no longer needed. This anomaly is one of the prevalent issues that damage program performance, and it is mostly located using memory profilers [52, 78, 139].

## 1.1 Analyzing Memory Usage

As mentioned, programmers locate memory anomalies through manual inspection of the code or using memory profilers. Identifying memory bugs by manual inspection is a complex and error-prone task since it requires the programmer to comprehend the program’s functionality and perform an exhaustive inspection of code. However, this knowledge is usually insufficient to locate and repair memory issues [52, 139]. Hence, to assist developers in this activity, software development environments provide memory profilers to monitor and report resource usage during software execution.

A profiler is designed to report information about the behavior exhibited by a target program during its execution. Profilers help developers evaluate how well software applications perform based on dynamic aspects, such as execution time, frequency of calls, and memory allocations. Therefore, memory profilers are tools that help programmers assess the memory usage of a program. Memory profilers collect a wide range of aspects from a software application and report information through full-text reports [80, 117], non-interactive visualizations [3, 6], and interactive visualizations [27, 54, 156] to support programmers in analyzing memory consumption.

Memory profilers provide different metrics such as garbage collection information (*e.g.*, [7, 117, 156, 157]), memory allocations (*e.g.*, [22, 27, 54, 98, 156, 157]), calling relationships between functions (*e.g.*, [5, 22, 98, 157]), among others. Consequently, this information helps programmers in analyzing certain aspects of the software application. For instance, consider Tracemalloc [7], a memory profiler tool provided by Python to trace memory blocks allocated using full-text reports. The Tracemalloc API helps users with a variety of activities, such as (i) tracing where an object was allocated, (ii) showing statistics about the allocated memory blocks per filename and per line number: total size, number, and the average size of allocated memory blocks, and (iii) computing the differences between two snapshots to detect memory leaks.

To exemplify the features of Tracemalloc, Figure 1.2 illustrates a piece of the traceback of the object that allocates most memory of the running example of Figure 1.1. This report indicates that most memory is allocated by the line where `to_json` function is called. If we scrutinize the traceback, we can notice that the execution path involves code lines from the `generic.py` and `base.py` files and are responsible for allocating most memory. These files are modified to repair the memory leak in the reported issue.

```

File "PandasIssue.py", line 6
  df.to_json(orient='split', indent=0)
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/core/generic.py", line 2305
  indent=indent,
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/io/json/_json.py", line 84
  indent=indent,
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/io/json/_json.py", line 144
  self.indent,
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/io/json/_json.py", line 244
  indent,
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/io/json/_json.py", line 166
  indent=indent,
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/core/indexes/base.py", line 3852
  return self._data.view(np.ndarray)
File "/opt/anaconda3/envs/vismepExperiment/lib/python3.7/site-packages/pandas/core/indexes/range.py", line 166
  self.start, self.stop, self.step, dtype=np.int64

```

Figure 1.2: The traceback of the object that allocates most memory of the running example of memory issue #45489 reported in the pandas package. The report shows the part of code and the execution path which allocate the most memory during program execution.

Although programmers are equipped with memory profiler tools in several programming languages (*e.g.*, Java, C++) [27, 54, 100, 113, 156, 172], programmers sometimes spend a considerable amount of time locating memory anomalies and repairing them using these tools since they struggle to find the required information or interpreting the report, among others [34, 52, 139, 166].

In this dissertation, we study how software visualizations support programmers in analyzing memory usage. Consequently, we describe in the next sections the characteristics of software visualization, how they are evaluated, and the drawbacks of software visualizations, considering the aspects previously mentioned.

## 1.2 Visualizing Memory Consumption

Software visualizations are known to be adequate at supporting practitioners in software comprehension and impacting the understanding of software analysis positively [43, 45, 103]. Thus, numerous software visualizations have been proposed over the years to facilitate the analysis of memory usage for programmers. As well as the memory profilers that provide textual reports, software visualizations display a broad spectrum of information (*e.g.*, object creation [82, 97], memory access [54, 118]) using diverse visual representation (*e.g.*, node-link diagrams [9, 22, 156], flame graphs [27, 54], sunburst [46]) and interaction mechanisms to help developers with memory consumption analysis. For instance, in 2018, we proposed a visual memory profiler [22] for Pharo that uses a graph with visual hints to display information about the calling relationships between methods, the objects allocations, the number and the type of objects allocated, and the code responsible for allocating these objects. This visualization helps programmers locate and analyze object allocation sites to detect unnecessary object allocations that can lead to memory growth or high workload in the garbage collector.

Past studies introduced and described the potential benefits that memory profilers can offer to analyze the memory consumption of applications [27, 54, 98, 157]. These studies usually show how their authors locate or repair memory issues reported in open-source projects. However, they do not assess whether these tools satisfy developers' information needs when analyzing memory usage. Furthermore, few studies present an evaluation showing detailed

information about how proposed software visualizations perform when programmers analyze memory usage and address memory anomalies [22, 63, 156]. Therefore, there is limited empirical evidence about (i) what information programmers need when analyzing memory and addressing issues, (ii) how programmers employ tools to satisfy these needs, and (iii) the user experience of programmers when employing tools.

### 1.3 Understanding the Problem of Visualizing Memory Usage

As mentioned, diverse software visualizations have been proposed to analyze memory consumption during software applications' development. Nonetheless, a number of open problems have limited their widespread use. We believe that these drawbacks are:

- *Difficulties finding an appropriate software visualization.* Software visualizations are designed considering several characteristics (*e.g.*, data, visual representation) to support developers when performing a task. A developer might struggle to find a suitable visualization for analyzing memory usage and addressing anomalies due to the lack of organization and information amongst proposed visualizations. Finally, if prototypes are unavailable, this affects (i) practitioners for employing adequate visualizations and (ii) researchers for comparing new proposals with the state-of-art or identifying potential challenges and hints for improvement.
- *Evidence of software visualizations in practice.* How practitioners perceive and interact with visualizations can significantly affect their understanding of data and the approach's usefulness [96, 144]. A limited number of articles present evidence about how practitioners employ visualizations during memory consumption analysis [22, 63, 156]. Consequently, little is known about how developers employ visualizations to satisfy their needs. Also, most proposals fail to examine if the problem domain of visualizations is in touch with the needs of developers, the data is sufficient and helpful, or the visual representation is intuitive, among others. Even if a proposed visualization is found and considered suitable, practitioners are usually unsure of the effectiveness of visualizations.
- *Information about programmers needs and challenges.* To the best of our knowledge, there is no detailed empirical data about what information is required by programmers when analyzing memory usage and addressing memory anomalies, how they analyze the information required to perform tasks and the challenges in which programmers need assistance. We consider this information valuable for designing suitable tools in this context and demonstrating their usefulness to practitioners.

In this dissertation, we focus on obtaining detailed information on the characteristics of the visualizations and how people currently use and perceive these tools to obtain information that meets their needs. Thus, our empirical foundation can provide valuable support to design tools that adequately help developers analyze memory consumption.

## 1.4 Our Proposal

We consider that software visualizations could be an alternative to support programmers analyzing memory usage. However, previously mentioned challenges limit tool builders and researchers in providing adequate tools and visual support to help developers assess their programs' memory consumption. Therefore, providing an empirical foundation based on specific aspects could be valuable for researchers and tool builders to design and improve tools that assist programmers with the correct problems in a way that is more likely helpful and suitable for practitioners. For our empirical foundation, we consider (i) what information programmers need when analyzing memory and addressing issues, (ii) how programmers employ tools to satisfy these needs, and (iii) the user experience of programmers when employing tools.

*Thesis statement.* We formally state our thesis as follows:

An empirical foundation based on the exploration of (i) what information about software aspects programmers need to understand, (ii) how they use tools to discover that information, and (iii) how they perceive the tools improves the design of software visualizations that facilitate programmers in analyzing memory usage.

### 1.4.1 Research goals

We have identified the following research goals:

- Understand (i) the main characteristics of current visual approaches that help programmers in analyzing memory usage, (ii) the challenges of the field, and (iii) research areas that are worth exploring.
- Propose and develop a software visualization tool to analyze applications' memory usage based on the characteristics of state-of-art.
- Propose and conduct an empirical study to understand how the implemented visual tool supports practitioners when analyzing memory usage and how they perceive the tool.
- Propose and conduct an empirical study to comprehend (i) what programmers need to know when analyzing memory usage and (ii) how programmers find that information using memory tools.

## 1.5 Contributions

The main contributions of this thesis are summarized as follows and have been published in the listed references:

- *Literature review.* We have systematically selected and categorized articles centered on software visualizations that support practitioners in analyzing the memory consumption of programs based on the (i) tasks supported, (ii) data abstracted, (iii) visual representation, (iv) evaluations conducted, and (v) prototype availability. We identified the main challenges of visualizing memory consumption and evaluating approaches, as well as, opportunities for improvement [23].
- *Software visualization in practice.* We implemented Vismep, an interactive visualization prototype to help programmers analyze Python applications' memory usage as shown in Figure 1.3. This prototype is designed based on the findings from the previous contribution and characteristics of state-of-art. In addition, we also conducted an exploratory study with programmers to validate how Vismep supports practitioners when analyzing memory usage [24]. Section 1.6 provides an overview about our visual approach.
- *Programmer needs and tool usage.* We conducted observational studies to understand (i) what a programmer needs to know when analyzing memory usage and (ii) how a programmer finds that information using memory tools. We provide a catalog of questions programmers ask themselves when analyzing memory consumption and addressing memory issues. Additionally, we presented a detailed analysis of how some tools are used to answer these questions (**Under review**).

## 1.6 Vismep Overview

One of the contributions of this dissertation is Vismep, an interactive visualization prototype to assist programmers in analyzing the memory usage of Python programs (see Chapter 3). This section briefly summarizes our proposed visualization through a small example and the issue mentioned before.

Vismep outlines how the program runs and uses memory during its execution using four views. The main view, called *Call graph view*, shows the program execution with a memory footprint through a call graph with visual hints. To illustrate, consider the code (a) in Figure 1.3 that generates a `Canvas`, adds 1000 shapes to the canvas, and then asks if the canvas is empty. However, to verify if the canvas is empty (`isEmpty` function), the `allShapes` function is called. Consequently, an unnecessary computational operation is made for this program goal as shown in code of Figure 1.3. The unnecessary allocation of objects is one of the reasons that cause memory bloats [156, 22]. Memory bloat is an issue that exposes the inefficient use of memory by a program [156]. Note that an application may be free of memory leaks but could require excessive memory to operate correctly due to memory bloat. Figure 1.3 shows the main view of Vismep in two different situations. We observe the main view of Vismep (b) for running the canvas example and the view (c) for the canvas code without unnecessary allocations. Note that the call graph presents fewer nodes in the view without the memory issue than in the original code.

**Vismep in action.** To show how Vismep performs with larger and more complex applications, consider the reported issue in Figure 1.1. Figure 1.4 displays on the right, the main view of

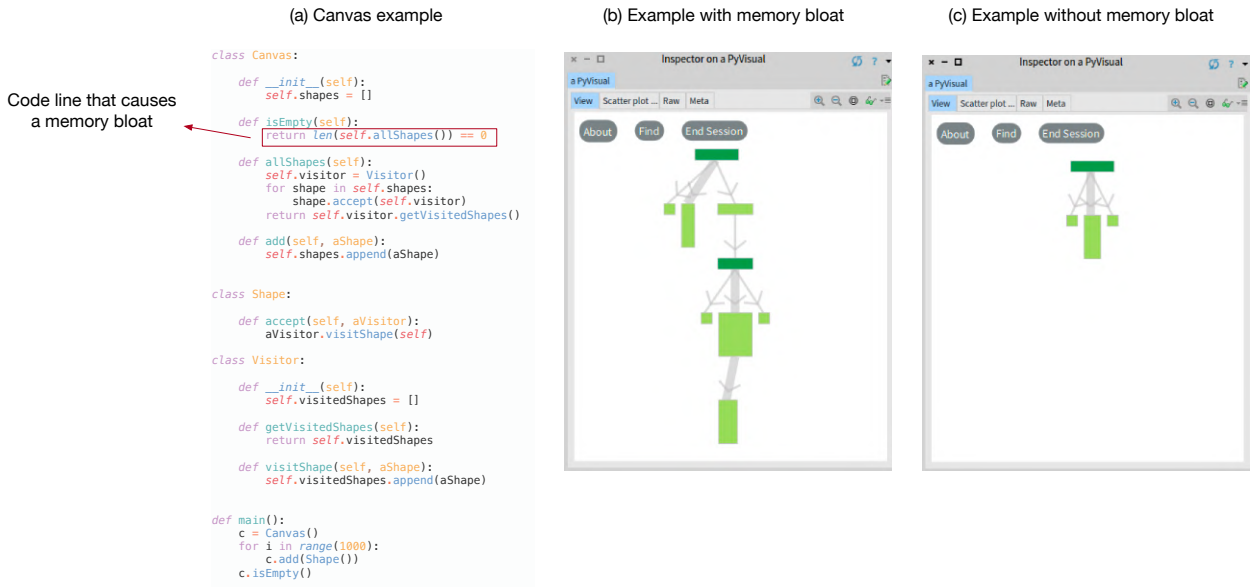


Figure 1.3: (a) A code example that illustrates a canvas with many shapes and performs an unnecessary computational operation to verify if the canvas is empty. (b) The main view of Vismep for running example of canvas, and (c) running example without the memory issue.

Vismep running example in Figure 1.1 consuming in total around 728 MB. On the left, we observe the view of Vismep without the memory issue in the running example, consuming a total of 299 MB. Note that both call graphs’ structures are similar, but they present some differences in the execution path of `to_json` function, which is responsible for allocating most memory. On the right, the execution path contains more method/function calls than on the left due to the changes to repair the memory issue.

## 1.7 Scope and Limitations

This dissertation voluntarily focuses on the Python ecosystem, and the conducted explorative studies involve programmers with experience in Python and different applications using popular Python packages. Python is frequently used and supported by the Python community and industry (<https://www.python.org/>) in diverse topics (*e.g.*, machine learning, data science). However, few studies focused on analyzing memory consumption and memory issue detection in Python. Most literature centered on memory usage analysis considers other languages (*e.g.*, Java, C++).

We believe this thesis provides valuable insights into (i) what information programmers look for to analyze memory usage and address memory issues, (ii) how programmers employ tools to get that information, and (iii) their perception of tools. However, we do not diversify our thesis by covering other programming languages, software applications, and tools due to the difficulty of conducting such studies on a large scale. Consequently, our findings may not be generalized to other software applications and memory profiling tools beyond the Python ecosystem.



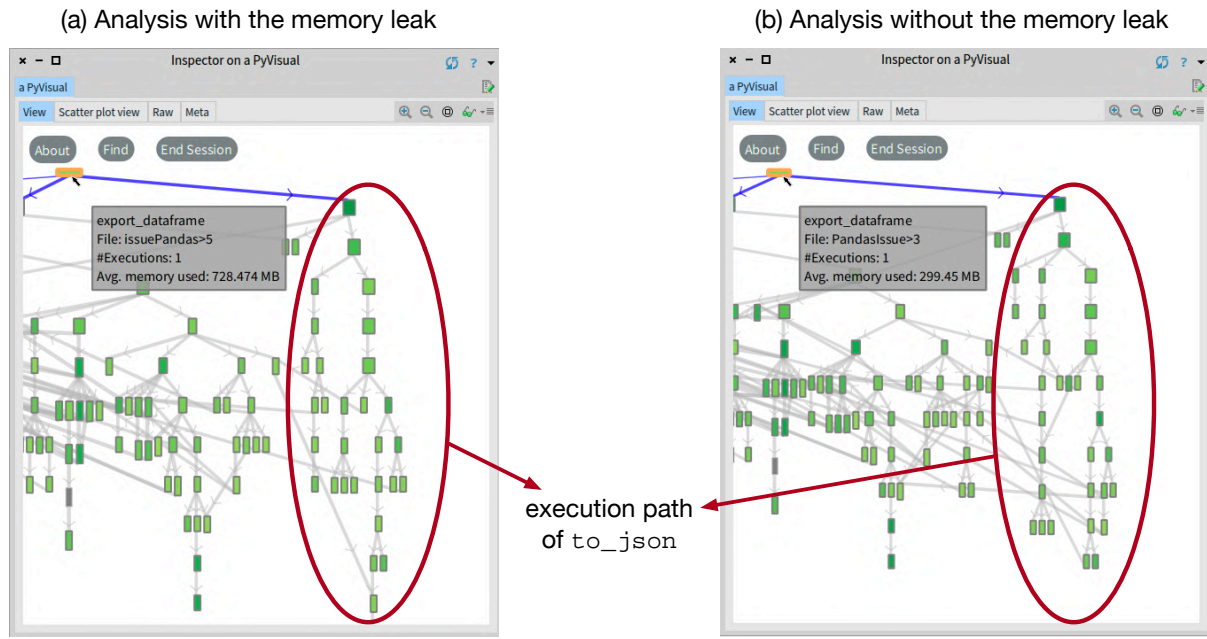


Figure 1.4: The main view of Vismep for running example in Figure 1.1 (right) and running example without the memory issue (left).

## 1.8 Related Publications

This section lists all publications and products created during the Ph.D. work.

**Main contributions.** The main contributions of this thesis have been published as follows:

- Alison Fernandez Blanco, Alexandre Bergel, and Juan Pablo Sandoval Alcocer. Software visualizations to analyze memory consumption: A literature review. *ACM Computing Surveys (CSUR)*, 2022, vol. 55, no 1, pp 1-34.
- Alison Fernandez Blanco, Alexandre Bergel, Juan Pablo Sandoval Alcocer, and Araceli Queirolo Cordova. Visualizing Memory Consumption with Vismep. *Proceedings of the 2022 IEEE Working Conference on Software Visualization (VISSOFT)*.
- Alison Fernandez Blanco, Araceli Queirolo Cordova, Alexandre Bergel, and Juan Pablo Sandoval Alcocer. Asking and Answering Questions During Memory Consumption Analysis. (**Under review**)

**Other contributions.** Tangentially related to this thesis, the author also co-authored:

- Alison Fernandez Blanco, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. Effective visualization of object allocation sites. *Proceedings of the 2018 IEEE Working Conference on Software Visualization (VISSOFT)*.
- Alison Fernandez Blanco. Towards memory consumption visualization for non-experts. Presented at *Doctoral Symposium ICSME 2020*.

*Software.* Software products developed during the author’s thesis:

- Alison Fernandez Blanco, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. Effective Visualization of Object Allocation Sites. Artifact presented at *Proceedings of the 2018 IEEE Working Conference on Software Visualization (VISSOFT)*. Available at: <https://doi.org/10.5281/zenodo.1311788>
- Alison Fernandez Blanco, Alexandre Bergel, Juan Pablo Sandoval Alcocer, and Araceli Queirolo Cordova. Visualizing Memory Consumption with Vismep. Available at: <https://github.com/Balison/Vismep>

## 1.9 Thesis Outline

This dissertation is structured as follows:

- **Chapter 2, Software Visualizations to Analyze Memory Consumption**, presents a systematic literature review of software visualizations that help analyze the memory consumption of programs. It introduces a taxonomy based on five dimensions and highlights (i) the main characteristics of current visual approaches, (ii) the challenges of the field, and (iii) a number of research areas that are worth exploring.
- **Chapter 3, Visualizing Memory Consumption with Vismep**, introduces Vismep. More precisely, it describes how we design Vismep based on the common characteristics used in the state-of-art and the areas that are worth of exploring. This software visualization tool that helps programmers analyze memory usage in Python applications. It also presents an exploratory study that illustrates how programmers employ Vismep to analyze the memory usage of Python applications and the perception that they have of Vismep.
- **Chapter 4, Answering and Asking Questions During Memory Consumption Analysis**, explores programmer questions during memory usage analysis when using Tracemalloc and Vismep. It provides a detailed analysis of how programmers use to answer those questions. It also highlights the barriers that programmers face during the process and discusses how well existing tools could support a programmer in answering these questions.
- **Chapter 5, Conclusions**, summarizes the contributions of this work and discusses the limitations and our future research directions.

# Chapter 2

## Software Visualizations to Analyze Memory Consumption

Understanding and optimizing memory usage of software applications is a difficult task, usually involving the analysis of large amounts of memory-related complex data. Over the years, numerous software visualizations have been proposed to help developers analyze the memory usage information of their programs. This chapter reports a systematic literature review of published works centered on software visualizations for analyzing the memory consumption of programs. We have systematically selected 46 articles and categorized them based on the (i) tasks supported, (ii) data abstracted, (iii) visual representation, (iv) evaluations conducted, and (v) prototype availability. As a result, we introduced a taxonomy based on these five dimensions to identify the main challenges of visualizing memory consumption and opportunities for improvement. Additionally, we describe a number of research areas that are worth exploring.

The content of this chapter is based on the publication “Software Visualizations to Analyze Memory Consumption: A Literature Review” [23] (co-authored with Alexandre Bergel and Juan Pablo Sandoval) and has been reformatted according to departmental guidelines.

### 2.1 Introduction

Software development often involves deep and intricate technical aspects. Execution time and memory consumption are two primary resources in software engineering [137, 139]. Keeping the amount of memory consumed by a software system under control is an example of such a programming challenge.

***Understanding software execution.*** Manually understanding and addressing memory issues is challenging since it usually involves analyzing several metrics at once and requires a thorough analysis of the respective code [22, 29]. Therefore, software development environments provide tools to monitor and report resource usage during software execution to support programmers in these activities. An example of such a tool is the execution profiler,

designed to report information about the behavior exhibited by a target program during its execution. The information reported is usually displayed through full-text reports or textual tables.

***Visualizing memory consumption.*** Over the years, the research community of software visualization has proposed a variety of visualizations to support software comprehension [45, 74]. It has also been shown that interactive visualization reinforces the cognition that facilitates human interaction to explore and understand data [146]. Due to this, software visualizations enriched with interaction mechanisms become a powerful alternative for displaying profiler reports to support developers in understanding and addressing memory-related issues. Nonetheless, the lack of organization and information around these visual approaches hinders their widespread use and limits the identification of potential challenges and hints for improvement in the field.

This chapter presents a systematic literature review of software visualizations to support programmers in analyzing memory consumption and addressing memory issues. We initially used keyword searches against three popular scientific databases and complemented it with a bi-directional snowballing and a manual search of relevant venues. As a result, we found 420 articles published without counting duplicates. We then selected 46 articles based on inclusion/exclusion criteria and quality assessment. In this way, we included only the studies centered on visualizations to analyze the memory consumption of a software program. Consequently, we excluded articles that only focus on memory issues without visualization, articles that analyze the memory used by the visualization per se, and articles that focus on other performance metrics excluding memory usage. In summary, our systematic review focuses on published works centered on visualizations that assist practitioners in examining memory usage to identify optimization opportunities.

***Structure of the chapter.*** Section 2.2 presents the methodology we followed in this study. Section 2.3 displays the main findings by answering the questions defined in Section 2.2.1. Section 2.4 provides the open challenges for the new visualizations centered on analyzing memory usage. Section 2.5 discusses the state-of-art. Section 2.6 discusses the threats to the validity of this study, and Section 2.7 exposes the future work and the conclusions.

## 2.2 Methodology

This literature review follows a systematic and rigorous methodology to identify and categorize literature related to memory consumption visualization. We use a seven high-level steps methodology inspired by well-recognized software engineering guidelines for systematic reviews [69, 70]. Our steps are shown in Figure 2.1.

We describe each one of these steps in the following sections.

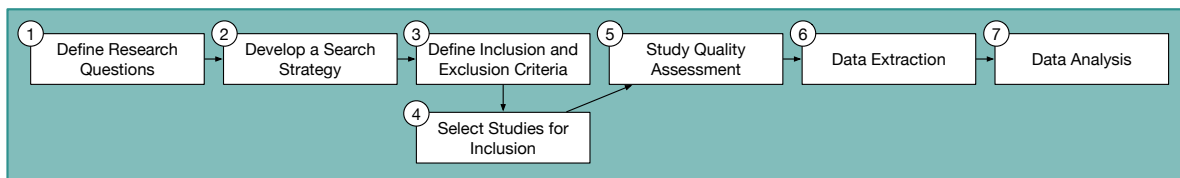


Figure 2.1: Overview of the workflow of the literature review.

## 2.2.1 Research Questions

The purpose of this literature review is to inspect, analyze, and discuss the state-of-art regarding software visualizations focused on helping developers to understand memory consumption. In particular, we are interested in addressing the research questions described in Table 2.1. We believe that answering these research questions (RQ) will assist future researchers in creating new visualizations focused on supporting developers during memory consumption analysis.

Table 2.1: Research Questions

Research Question	Dimension & Rationale
<b>RQ1:</b> <i>Which tasks are supported by the software visualizations to help users with the analysis of memory consumption?</i>	<i>Problem Domain:</i> Identify the tasks that software visualization targets to facilitate during the memory consumption analysis. For instance, identify bottlenecks or detect memory leaks.
<b>RQ2:</b> <i>What aspects of the software are abstracted by the software visualizations to help users with the analysis of memory consumption?</i>	<i>Data:</i> Software visualizations display large amounts of data (e.g., memory allocations, memory accesses) extracted from the execution or code of software applications. This information allows developers to understand the memory consumption of a program.
<b>RQ3:</b> <i>Which software visualizations have been proposed to help users with the analysis of memory consumption?</i>	<i>Visual Representation:</i> The use of different visual techniques to abstract complex and related data is an important topic. The way on which visual elements are rendered and presented to the user is also relevant because it may impact how the user interacts and perceives the visualization. In particular, we are interested in reviewing: <b>RQ3.1:</b> <i>Which visual techniques are used?</i> <b>RQ3.2:</b> <i>Which interaction tasks are supported?</i> <b>RQ3.3:</b> <i>Where are the visual elements rendered?</i>
<b>RQ4:</b> <i>How are software visualizations to help users with the analysis of memory consumption evaluated?</i>	<i>Evaluation:</i> Analyzing how software visualization is evaluated provides (i) an overview of the proposed visualization’s effectiveness and usefulness and (ii) a better understanding of conducted evaluation strategies.
<b>RQ5:</b> <i>What software visualization tools or prototypes are available to help users with the analysis of memory consumption?</i>	<i>Availability:</i> The availability of a prototype or tool is an opportunity (i) for practitioners to benefit from the approach and (ii) for researchers to replicate the results or complementing the associated research articles.

**Dimensions.** Our research questions focus on five dimensions: problem domain, data, visual representation, evaluation, and artifact. The research questions and their dimensions were inspired by six surveys [81, 83, 85, 105, 106, 119]. These studies present a number of relevant dimensions to give an enriched overview of software visualizations. Table 2.2 shows the six surveys mentioned previously with their respective dimensions and how they are related to our research questions.

Table 2.2: An overview to the relations between our dimensions and the dimensions proposed by some works of the state-of-art.

Survey	RQ1	RQ2	RQ3	RQ4	RQ5
Price <i>et al.</i> [106, 105]	Purpose	Scope, content	Form, method, interaction, effectiveness	Empirical evaluation	-
Roman <i>et al.</i> [119]	-	Scope, abstraction	Specification, method, interface, presentation	-	-
Maletic <i>et al.</i> [81]	Task	Target	Representation, medium	-	-
Mattila <i>et al.</i> [83]	Context	Data source	Visualization aspects	Evaluation aspects	-
Merino <i>et al.</i> [85]	Task	Data source	Representation, medium	-	Tool

**RQ1** centers on software engineering tasks supported by the visualization. **RQ1** was inspired by three previous studies: Price *et al.* [105, 106] provide taxonomies with a minor summary about the intention of the visualizations on their *Purpose* dimension. Maletic *et al.* [81], Mattila *et al.* [83] and Merino *et al.* [85] consider general tasks of software engineering like reverse engineering, maintenance, and testing. Compared to these works, our *Problem Domain* dimension focuses on detailed software engineering tasks related to memory usage.

In the case of **RQ2** and **RQ3**, the surveys mentioned previously present detailed information for these dimensions, providing an analysis of the collected data and how this data is abstracted visually to the user. In this literature review, our *Data* dimension describes the metrics considered for the analysis of memory usage, and the *Visual Representation* dimension reports the visual encodings, interactions, and medium used by the approaches.

Our study also includes two dimensions: **RQ4** and **RQ5** corresponding to evaluation and availability. **RQ4** was only covered by Price *et al.* [105, 106] and Mattila *et al.* [83], and **RQ5** by Merino *et al.* [85]. We include both dimensions since they are relevant in the research community to understand how the visualizations were evaluated and if they may be replicable.

## 2.2.2 Search Strategy

**Initial manual search.** According to the Systematic Literature Review guidelines [70, 162], before performing an automatic search phrase and defining an inclusion/exclusion criteria, it is necessary to search for an initial set of relevant articles. To do this, we manually reviewed the articles published between 2017 and 2020 in the following scientific venues:

- *IEEE International Working Conference on Software Visualization (VISSOFT)*
- *International Symposium on Memory Management (ISMM)*

We selected these conferences because the articles dedicated to software visualization and memory management present a sound corpus for our study. Besides, these conferences are classified respectively in the good (B) and excellent (A) category according to CORE rankings<sup>1</sup>, which determines conference rankings based on a mix of indicators (*e.g.*, citation rates, paper submission, acceptance rates). The result of our initial manual search ends up with five articles [22, 27, 54, 157, 123]. We used these papers as a base to define our search strategy by extracting search terms derived from the research questions.

***Search phrase development.*** We extracted the search terms that fit our scope of the title, abstract, and keywords from the articles found at the initial manual search. Furthermore, we expanded these search terms with synonyms and alternatives as shown in Table 2.3.

Table 2.3: Search terms and alternatives of spelling

Term	Alternatives
Memory*	memory heap, memory allocation, memory consume, memory consumption, memory usage, memory management, memory issues, memory issue, memory bloats, memory leaks, memory access, memory address
Visual*	visualize, visualization, visualisation, visual, visuals, visualizations, visualisations
Software*	software, program, application

To find potential articles considered for our study, we combined these terms into a query as it follows:

*Memory\* AND Visual\* AND Software\**

The previous query represents the condition that an article should meet to be considered in our study. We executed the query against the *abstract*. We did not limit the search based on publication date to find the most significant number of relevant articles for our study. We performed the search over three digital libraries:

- *ACM Digital Library*
- *IEEE Xplore*
- *Scopus*

---

<sup>1</sup><https://www.core.edu.au/conference-portal>

As a result, we found 533 papers that meet these criteria, including our initial set of five papers. The latter gives a level of certainty that we could find any article that proposes a visualization to assist developers with memory usage analysis. However, we may have a number of false positives that we detected in the following steps. Additionally, annex A presents the search strings used for each digital library mentioned before.

***Additional manual paper selection.*** In the previous phase, we found articles that contain the keywords used in the query search. As a result, we located articles that may be useful and representative. However, we may have missed some relevant articles. For instance, articles that use more particular memory-related keywords (*e.g.*, cache, fragmentation) may or may not be considered by our query. Therefore, in order to not miss any related paper, we also performed a manual search on the last ten editions (2010-2020) from the following venues:

- *IEEE International Working Conference on Software Visualization (VISSOFT), the continue of IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT) and ACM Symposia on Software Visualization (SOFTVIS)*
- *International Symposium on Memory Management (ISMM)*

We selected these conferences because we noticed that most of the articles resulting from our automatic phrase search were published in them. We also reviewed only articles published in the last ten issues due to our time and human resources. In total, around 305 articles were published in these venues over the last ten editions. We manually reviewed each article based on its title and abstract. Consequently, we found ten articles that fall within the scope of this literature review. However, seven articles were found in the earlier phases. Therefore, we identified three additional articles during this phase.

***Bi-directional snowballing.*** We performed a backward and forward snowballing over the ten articles found in the previous phase to complete our search. The snowballing procedure consists of identifying additional studies using the system of references between articles [161]. For this reason, we checked the references in each article, and we reviewed the list of articles that reference any article of our selection. Thus, we could add relevant research published after or before the publication date of our selection set by performing several iterations until non-relevant papers are referenced. We then selected *Google Scholar* to perform the forward snowballing due to the facilities provided to select the papers that cite a specific one. On the other hand, the backward snowballing was performed manually. Consequently, over two iterations, we found 56 additional articles that could be considered in this study, collecting a total of 420 papers without counting duplicated articles.

### 2.2.3 Inclusion & Exclusion Criteria

We elaborated inclusion and exclusion criteria based on the scope of this study. Table 2.4 details the inclusion and exclusion criteria. In particular, we are interested in papers that use visualization techniques to help developers understand and address memory issues.



Table 2.4: Inclusion and Exclusion Criteria

---

**Inclusion Criteria**


---

- *I1*: Papers published in a peer reviewed journal, conference or workshop on data visualization, computer science, or computer engineering.
- *I2*: Papers written in English.

---



---

**Exclusion Criteria**


---

- *E1*: Papers that focus on other performance metrics (*e.g.*, execution time).
  - *E2*: Papers that only study memory issues *or* visualization issues.
  - *E3*: Posters, keynotes, challenges and previous papers that only introduce the idea of most recent full papers (*e.g.*, short papers).
- 

We performed a revision of 420 articles based on inclusion/exclusion criteria. The thesis author and the two thesis supervisors independently read and analyzed the title, abstract, keywords, and venue to decide if an article is excluded or not. However, if a reviewer does not have enough information to decide, the reviewer should read the introduction and conclusions of the article. Next, each reviewer responds independently if an article should be included or not using a spreadsheet that lists the 420 articles.

Then, we examined the spreadsheet responses to calculate the Fleiss' kappa for the inter-rater reliability [49]. As a result, we got 0.72 for the Fleiss' kappa analysis, which is generally considered a good agreement beyond chance [50]. We also identified 38 articles on which we have discrepancies in the spreadsheet responses. Most of these differences were related to E1 and E2 criteria. For instance, some articles focus on using a software visualization to understand the trace execution of programs, but not explicitly center on memory consumption. On the other hand, other articles are dedicated to analyzing memory problems, but not primarily with software visualizations.

To resolve all conflicts, we conducted a second review of the 38 articles, analyzing the full content of each article. We then had a discussion session to develop an agreement based on the responses from the second review. As a result, a total of 49 articles are candidates to be included in our study.

## 2.2.4 Quality Assessment

This phase involves the selection of the papers based on their quality [70, 104]. To exclude the articles with insufficient information to contribute to this study, we examine the theoretical contribution, and the experimental evaluation with the checklist used in software engineering surveys [102, 147] detailed in Table 2.5.

In this step, we assess the quality of each paper based on the checklist illustrated in

Table 2.5: Quality assessment adopted from [147]

#	Questions
<b>Theoretical contribution</b>	
1	Is at least one of the research questions addressed?
2	Was the study designed to address some of the research questions?
3	Is a problem description for the research explicitly provided?
4	Is the problem description for the research supported by references to other work?
5	Are the contributions of the research clearly described?
6	Are the assumptions, if any, clearly stated?
7	Is there sufficient evidence to support the claims of the research?
<b>Experimental evaluation</b>	
8	Is the research design, or the way the research was organized, clearly described?
9	Is a prototype, simulation or empirical study presented?
10	Is the experimental setup clearly described?
11	Are results from multiple different experiments included?
12	Are results from multiple runs of each experiment included?
13	Are the experimental results compared with other approaches?
14	Are negative results, if any, presented?
15	Is the statistical significance of the results assessed?
16	Are the limitations or threats to validity clearly stated?
17	Are the links between data, interpretation and conclusions clear?

Table 2.5. We assigned a score to every question in the checklist independently. The score has a numeric scale of three levels: yes (2 points), partial (1 point), and no (0 points). The final score of a paper is measured by summing up the score of all questions. Since the form has 17 questions, the total score of the articles varies from 0 to 34.

Additionally, we follow the criteria of Usman *et al.* [147] by using the lower quartile ( $34/4 = 8.5$ ) as the limit point for including an article based on quality. As a result, all the articles with a score above 8.5 points were considered relevant hence they present enough information to address our research questions.

In total, 35 articles met the quality assessment with the approval of the thesis author and the two thesis supervisors, while 14 articles were detected as discrepancies. Consequently, a second pass was made over these 14 articles. At the second pass, each reviewer independently read and examined the quality assessment of each article again. We then moved on to a discussion session to resolve conflicts by consensus. Finally, with the second pass, a total of 46 articles were selected to be included in the literature review.

## 2.2.5 Data Extraction

To extract the necessary data, the thesis author was in charge of examining each of the 46 articles. From each article, the thesis author collected general information (*e.g.*, title, publication year, venue) and information according to the dimensions and rationale of the research questions. Although the data extractor reviewed the entire document, the data extractor focused on many particular sections in order to answer the research questions:

- *RQ1 – Problem Domain:* Abstract, introduction, evaluation, conclusion.
- *RQ2 – Data:* Data collection, data extraction, profiling information.
- *RQ3 – Visual Representation:* Visualization, detailed view, visual design, display.
- *RQ4 – Evaluation:* Evaluation, case study, applications, usage scenario.
- *RQ5 – Availability:* Visualization, implementation, conclusion.

The data extractor was also careful to search for data to respond to *RQ5* because sometimes artifacts or data sets are placed as a reference or footnotes.

In order to validate the data extraction, the two thesis supervisors checked the data to confirm that extraction was correct. Next, we discussed and resolved any disagreements by reviewing the articles and data extraction forms. We then recorded the final data value for data analysis.

We noticed that some articles do not present information to respond to all the research questions during this phase. For example, some articles lack information about the interactions supported, the medium used, or the evaluation conducted. We discussed the data synthesis of these cases in Section 2.2.6 and Section 2.3.

## 2.2.6 Data Analysis

This section describes the data analysis methods conducted to answer our research questions.

***Thematic analysis.*** We opted to conduct a thematic analysis [141] for RQ1 and RQ2 since we noted that the proposed classification schemes from previous software visualization surveys were general for helping us answer these research questions. In order to create a classification scheme, the thesis author conducted the thematic analysis following a number of specific steps:

- *Familiarization.* Extracted data is read and reread to have an overview of the information.
- *Generating codes.* The thesis author assigned codes that reflect relevant features to answer the research questions. For example, the author assigned the code “*Detection of memory fragmentation*” for the text: “To help the user find potential memory fragmentation problems, we display memory blocks that have been freed exactly one time and not reused” [118]. Additionally, continuous reviews were conducted to refine codes and determine if they were assigned correctly. The latter requires comparing two text segments assigned to the same code to inspect if they reflect the same feature.
- *Constructing initial themes.* All codes are compiled with their associated data into coherent groups to identify initial themes (broader patterns) that help address the respective research questions. The codes that seem to not belong to a specific theme were grouped as miscellaneous and analyzed in the next step.

- *Reviewing themes.* Initial themes were checked against the associated data (*e.g.*, segments of text) and refined to create a final set of themes.
- *Defining and naming themes.* Each theme of the final set was defined with a detailed description and an informative name. For instance, the themes for RQ2 are generated based on the source of the data abstracted (*e.g.*, data from program execution, data from source code, data from versions).

Finally, the two thesis supervisors checked the process by reviewing the consistency of codes and themes against the associated data and examining if the themes created respond to RQ1 and RQ2. We held three meetings to discuss the disagreements or potential issues of the generated codes and themes. As a consequence, we minimized potential inconsistencies in the coding process.

**Content analysis.** We conducted a deductive content analysis [41, 47] to answer RQ3.1, RQ3.2, and RQ4 since the data synthesis was performed based on defined classification schemes from previous studies shown in Table 2.6.

Table 2.6: Classification scheme

ID	Dimension	Proposed by	Classification scheme
RQ3.1	Visual techniques	Keim [66]	Geometrically-transformed displays, iconic displays, dense pixel displays, stacked displays and standard 2D/3D displays
RQ3.2	Interactions	Yi <i>et al.</i> [167]	Select, explore, reconfigure, encode, abstract/elaborate, filter and connect
RQ4	Evaluation	Merino <i>et al.</i> [84]	No explicit evaluation, empirical, theoretical

For RQ3.2, we classified only the articles that present information to answer the research question. We exposed the number of articles that lack data to answer these research questions. For RQ4, Merino and colleagues proposed a category of “No explicit evaluation” for these cases.

The classification of articles for RQ3.1, RQ3.2, and RQ4 was performed by the thesis author and one thesis supervisor independently. Each one filled a spreadsheet to classify the articles based on a detailed description of the predetermined categories. Later, to check the agreement between reviewers, we calculated metrics for reliability (Cohen’s kappa [152], percentage of agreement). As a result, we noticed that reviewers present a “substantial agreement” ( $kappa > 0.61$ ) and a percentage agreement above 80% at classifying the articles for most categories. However, we noted disagreements on the classification based on the interactions supported and the evaluations conducted (case study vs. usage scenario). We discussed the disagreements in meetings by exposing the data and examining the description of the categories. Consequently, we resolved the discrepancies and advanced to expose the results to answer the research questions.

Finally, to respond to RQ3.3 we only reviewed the medium employed. And for RQ5, we listed the link for the prototype and the additional information (video, sample data) provided in the link.

## 2.3 Results

Table 2.7 summarizes the results of all steps in our systematic search methodology. It shows different stages of the process for search and selected relevant articles. *Unique* columns show the number of non-duplicated articles. As a result, we found that 32.27% of the articles found in the search phase over digital libraries were duplicated articles.

We also noticed that 11.66% of the articles found during the search phases were selected based on the inclusion/exclusion criteria.

Table 2.7: Systematic Search Results

Source	Date	Search Results	Unique	Incl./Excl. Criteria	Quality Assessment	Included
ACM DL	March 18, 2021	209				
IEEE	March 18, 2021	72				
SCOPUS	March 18, 2021	252				
Search phrase		533	361	27	24	24
Additional manual search		3	3	3	3	3
Bi-directional snowballing		56	56	19	19	19
					<b>Total</b>	46

In the end, Table 2.8 and Table 2.9 display the 46 articles that passed our inclusion/exclusion criteria and satisfied the criteria of our quality assessment. A set of collected data from the 46 articles is available online<sup>2</sup>.

**Publication year.** During the search phase in digital libraries, we do not limit the search based on publication dates. We also do not exclude articles based on their publication date during the selection phase. However, we noted that the articles considered by our study were published between 1996 and 2020 (see Figure 2.2). Furthermore, we detected that the number of published articles increased over time, with high peaks (4 articles published) in 2002, 2010, and 2018. Note that these peaks may be due to external factors such as adopting new paradigms or developing applications for new domains.

**Venues.** Regarding the distribution of articles based on the venue, we identified 27 different venues where the papers were published. Most of the venues are related to software visualizations, software maintenance and software comprehension. Furthermore, we observed that 32.60% of selected studies were published in software visualization conferences (VISSOFT and SOFTVIS). We also noticed that 17.39% of the articles were published in journals usually involved with computer graphics and visualizations (*e.g.*, Computer Graphics Forum, IEEE TVCG). Finally, the remaining articles were published in various conferences and workshops, usually related to software maintenance and software comprehension (*e.g.*, ISMM, ICSME).

<sup>2</sup><https://www.dropbox.com/s/7srvxiacftg2pm1/ArticleClassification.csv?dl=0>

Table 2.8: The included papers in the study (S1-S30)

<b>ID</b>	<b>Title</b>	<b>Venue</b>	<b>Year</b>	<b>Ref.</b>
S1	Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study	<i>PACMHCI</i>	2020	[156]
S2	Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor	<i>VISSOFT</i>	2020	[157]
S3	PVC.js: visualizing C programs on web browsers for novices	<i>Helijon</i>	2020	[63]
S4	Enhancing Commit Graphs with Visual Runtime Clues	<i>VISSOFT</i>	2019	[123]
S5	Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms	<i>CCGRID</i>	2019	[98]
S6	Detailed heap profiling	<i>ISMM</i>	2018	[27]
S7	Effective visualization of object allocation sites	<i>VISSOFT</i>	2018	[22]
S8	NumaMMA: NUMA MeMory Analyzer	<i>ICPP</i>	2018	[145]
S9	Memaxes: Visualization and analytics for characterizing complex memory performance behaviors	<i>TVCG</i>	2018	[53]
S10	Atlantis: Improving the analysis and visualization of large assembly execution traces	<i>ICSME</i>	2017	[61]
S11	Visual exploration of memory traces and call stacks	<i>VISSOFT</i>	2017	[54]
S12	Leveraging Managed Runtime Systems to Build, Analyze, and Optimize Memory Graphs	<i>VEE</i>	2016	[135]
S13	Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems	<i>ISPASS</i>	2016	[44]
S14	Efficiently identifying object production sites	<i>SANER</i>	2015	[62]
S15	TABARNAC: Visualizing and resolving memory access issues on NUMA architectures	<i>VPA</i>	2015	[18]
S16	Visualization of memory access behavior on hierarchical NUMA architectures	<i>VPA</i>	2014	[160]
S17	A visual approach to investigating shared and global memory behavior of CUDA kernels	<i>Comput Graph Forum</i>	2013	[120]
S18	Visualizing the allocation and death of objects	<i>VISSOFT</i>	2013	[151]
S19	Abstracting runtime heaps for program understanding	<i>IEEE TSE</i>	2012	[82]
S20	Topological analysis and visualization of cyclical behavior in memory reference traces	<i>PacificVis</i>	2012	[33]
S21	Vasco: A visual approach to explore object churn in framework-intensive applications	<i>ICSM</i>	2012	[46]
S22	Abstract visualization of runtime memory behavior	<i>VISSOFT</i>	2011	[32]
S23	A map of the heap: Revealing design abstractions in runtime structures	<i>SOFTVIS</i>	2010	[97]
S24	Allocray: Memory allocation visualization for unmanaged languages	<i>SOFTVIS</i>	2010	[118]
S25	Automated construction of memory diagrams for program comprehension	<i>ACM SE</i>	2010	[37]
S26	Heapviz: interactive heap visualization for program understanding and debugging	<i>SOFTVIS</i>	2010	[9]
S27	Making Sense of Large Heaps	<i>ECOOP</i>	2009	[89]
S28	Visualizing the Java heap to detect memory problems	<i>VISSOFT</i>	2009	[115]
S29	Hdpv: Interactive, faithful, in-vivo runtime state visualization for C/C++ and Java	<i>SOFTVIS</i>	2008	[138]
S30	Interactive Visualization for Memory Reference Traces	<i>Comput Graph Forum</i>	2008	[31]

Table 2.9: The included papers in the study (S31-S46)

ID	Title	Venue	Year	Ref.
S31	Visualizing dynamic memory allocations	<i>VISSOFT</i>	2007	[93]
S32	Visualising dynamic memory allocators	<i>ISMM</i>	2006	[28]
S33	Jove: Java as it happens	<i>SOFTVIS</i>	2005	[116]
S34	YACO: A User Conducted Visualization Tool for Supporting Cache Optimization	<i>HPCC</i>	2005	[111]
S35	Interactive locality optimization on numa architectures	<i>SOFTVIS</i>	2003	[94]
S36	Visualizing Java in action	<i>SOFTVIS</i>	2003	[114]
S37	GCspy: an adaptable heap visualisation framework	<i>OOPSLA</i>	2002	[109]
S38	Visualising the train garbage collector	<i>ISMM</i>	2002	[108]
S39	Visualizing memory graphs	<i>Software Visualization</i>	2002	[171]
S40	Visualizing the execution of Java programs	<i>Software Visualization</i>	2002	[39]
S41	Visualizing the impact of the cache on program execution	<i>ICIV</i>	2001	[168]
S42	Visualizing the memory access behavior of shared memory applications on NUMA architectures	<i>ICCS</i>	2001	[140]
S43	Visualizing reference patterns for solving memory leaks in Java	<i>ECOOP</i>	1999	[40]
S44	A cache visualization tool	<i>Computer vol. 30</i>	1997	[148]
S45	DDD — a free graphical front-end for Unix debuggers	<i>SIGPLAN Not.</i>	1996	[170]
S46	Monitoring data-structure evolution in distributed message-passing programs	<i>HICSS</i>	1996	[124]

### 2.3.1 RQ1: Problems Domain

Selected articles propose software visualizations that usually target to help developers perform debugging and performance tasks. We performed thematic analysis to find patterns over the data to provide details of which tasks are supported by these visualizations. As a result, we detected themes that help users adopt a suitable software visualization according to their requirements. We classified the visualizations based on (i) focus point analysis and (ii) issue detection. Table 2.10 shows the distribution of papers based on this classification. According to our classification, a visualization could focus on analyzing a specific point and detecting multiple memory issues. As a result, a visualization could belong to multiple categories.

**Focus point analysis.** Articles explain why the proposed visualization is helpful in different sections. During our thematic analysis, we noticed that a number of articles present a general description by specifying that the proposed visualization has a general purpose in helping developers understand and monitor an application’s memory consumption. On the other hand, we found articles that propose dedicated visualizations that allow users to analyze specific points (*e.g.*, data structure, cache behavior). We classified the articles based on the focus point analysis described below.

- *Specific architecture.* We found 23.91% articles dedicated to analyzing the memory consumption of applications with specific architectures (HPC, parallel, embedded, distributed). These articles usually propose visualizations that display how the data is

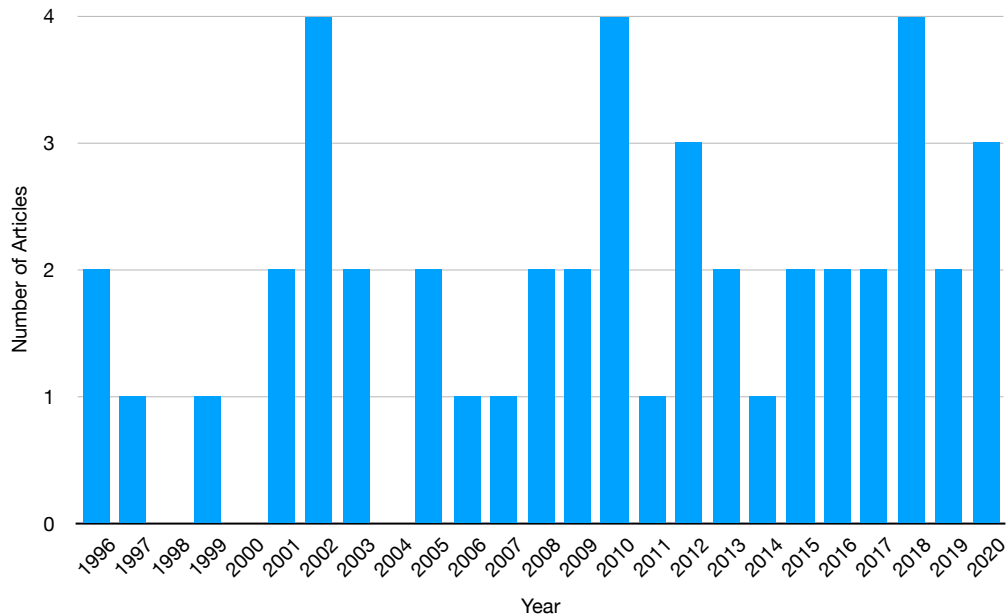


Figure 2.2: The 46 included papers by publication year.

Table 2.10: Classification of articles based on the tasks

Problem domain		References	Total
Focus point analysis	Specific architectures	S5, S8-S9, S12-S13, S15-S17, S35, S42, S46	11
	Data structure	S3, S19, S23, S25-S29, S33, S36, S39-S40, S43, S45	14
	Cache performance	S22, S30, S34, S41, S44	5
	Memory regression	S4	1
	General	S1-S2, S6-S7, S10-S11, S14, S18, S20-S21, S24, S31-S32, S37-S38	15
Issue detection	Memory leak	S1, S2, S24, S26-S29, S40, S43, S46	10
	Memory bloat	S1, S4, S6-S7, S14, S18-S19, S27-S28	9
	Memory churn	S1, S21, S24, S28-S29, S38	6
	Memory fragmentation	S5, S24, S31-S32, S37-S38	6

accessed and used by multiple threads and multiple processors. For example, article **S35** [94] allow developers to understand the memory access behavior of parallel NUMA applications. This article proposes a visualization that helps developers identify which nodes perform the memory accesses and detect an opportunity to reduce remote memory accesses.

- *Data structure.* According to Cormen *et al.* [36], a data structure is a way to store, manage and organize data to facilitate access and modifications. There is a variety of data structures employed in software applications (*e.g.*, lists, dictionaries). However, the inefficient usage of data structure and its operations (*e.g.*, adding, removing elements) generates memory issues that affect the performance. For this reason, data structure analysis is a prevalent task during software development. We detected that 30.46%



of visualizations support developers in analyzing and inspecting data structures. For instance, article **S26** [9] proposes *Heapviz* that allows developers to identify large data structures and which objects are shared by several data structures. *Heapviz* displays a node-link diagram to visualize the references between objects and locate the nodes using a radial layout to use screen space efficiently.

- *Cache performance.* Cache stores data so that future requests for that data can be responded more quickly. Tracking the cache activity in a software application helps developers understand memory performance at a fine-grained level. Accordingly, developers may require and analyze memory access and cache performance; hence, the cache activity's analysis influences detecting memory access anomalies. We found that 10.86% of visualizations support developers in analyzing cache performance. For example, article **S34** [111] propose *YACO* to help users with the analysis of access patterns and cache misses. *YACO* present multiple views to display statistics related to cache performance and allow developers to find data that frequently enter and leave the cache.
- *Memory regression.* Source code changes may impact the performance of an application [11]. Only article **S4** [123] allows developers to analyze the memory variations between code changes. This article proposes *Spark Circle* that enables users to compare two commits based on the number of allocated objects, the execution time, and the number of modified methods. As a consequence, a developer can identify the growth or reduction of allocated objects between commits.
- *General.* As we mentioned before, we found 32.60% of the articles do not focus on a specific point. These articles determine that the goal of the proposed visualization is to analyze memory consumption. Therefore we could not find specific analysis points such as analysis of data structures or cache behavior. During the generation of codes, we detected that these articles usually display memory access or heap usage. Furthermore, most of these articles are useful for detecting memory issues.

**Issue detection.** We found that 47.82% of the articles propose visualizations that allow developers to identify memory issues. Additionally, we detected several articles that claim to present helpful visualizations to address memory anomalies but do not specify which kind of anomalies or particular situations can be addressed with the proposed visualization. Consequently, we only classified the articles that present a detailed description of the memory issues addressed and how developers could employ the visualization to find these anomalies. We described the memory issues that were found below.

- *Memory leak.* A memory leak is an issue deeply related to improper memory management [25, 52]. A memory leak occurs when an unused memory allocation cannot be released from memory [40]. The latter usually happens because there are allocations that reference an unused allocation. As a consequence, the application can run out of memory and crash. We found that 21.73% of the visualizations help developers in detecting memory leaks. For instance, article **S40** [39] presents a node-link diagram to display the references between the objects not reclaimed by the garbage collector. This

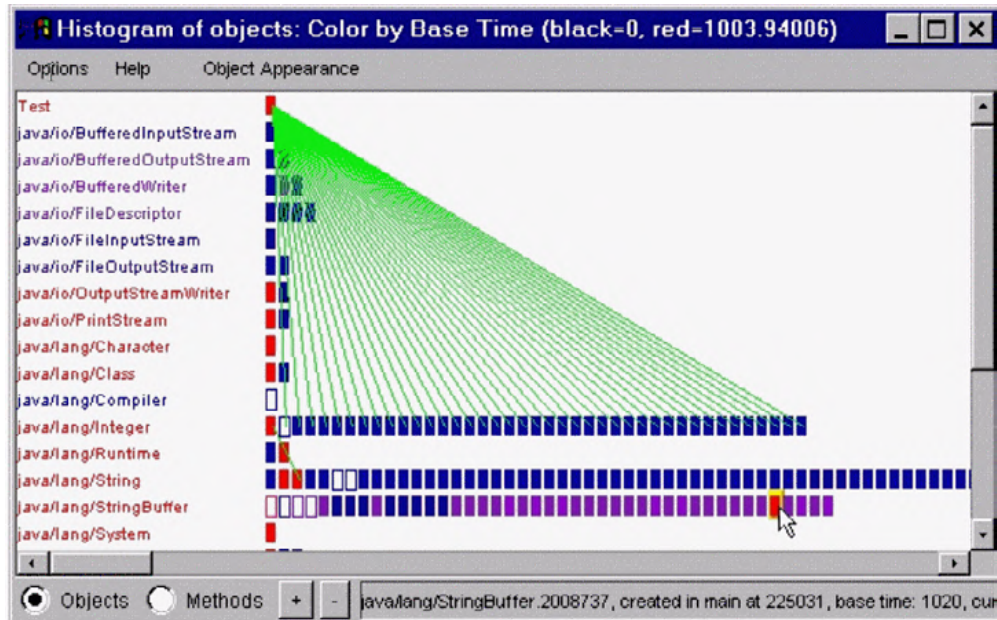


Figure 2.3: Visualization proposed by De Pauw *et al.* [39] to illustrate the references between the objects not reclaimed by the garbage collector in Java. ©2002 “Visualizing the Execution of Java Programs” by Wim De Pauw *et al.*. Reproduced with permission from Springer Nature.

visualization allows developers to identify memory leaks by exploring which objects are no longer used but are referenced by other objects (see Figure 2.3).

- *Memory bloat.* Memory bloat exposes inefficient use of memory by a program [156]. A memory overhead significantly affects software applications by reducing their scalability and usability. Developers should notice that an application may be free of memory leaks, but could require excessive memory to operate correctly. According to LaToza *et al.* [76], developers usually ask, “*How big is this in memory?*” and “*How many of these objects get created?*” These questions are related to distinguish memory growth. Addressing memory bloats impacts the application behavior, making it more usable and faster in some cases [65]. We detected that 19.56% of visualizations support developers in identifying excessive memory consumption. For example, article **S1** [156] allows developers to explore and observe memory consumption over time by using multiple views with *AntTracks*. As a result, *AntTracks* assists developers in detecting memory growth and exploring suspicious time windows in which objects are allocated over time, as shown in Figure 2.4.
- *Memory churn.* This issue occurs when an application allocates and releases a large number of short-living objects [46]. For example, memory churn can happen if a program allocates several new objects in the middle of nested loops. As a result, the time spent on allocating objects in a heap and the number of garbage collections increase. Thus the application decreases its performance due to frequent garbage collection. We detected that 13.04% of the visualizations help developers in detecting memory churn. For example, article **S21** [46] proposes *Vasco*, a visualization that allows developers to identify where and when an object is allocated and no longer used. The authors of

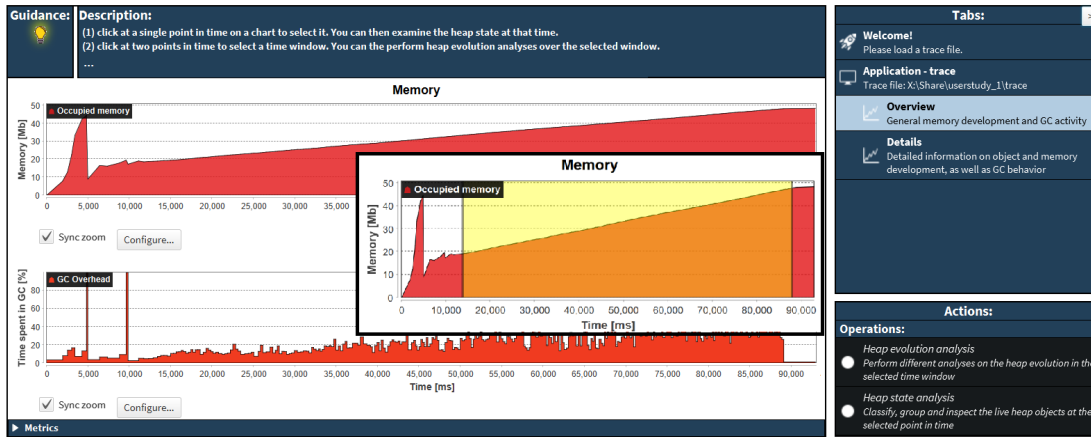


Figure 2.4: AntTracks [156] provides overview plots about the application’s memory footprint and garbage collector overhead and allows practitioners to explore a time window. ©Used with permission of ACM (Association for Computing Machinery), from “Evaluating an Interactive Memory Analysis Tool: Findings from a Cognitive Walkthrough and a User Study” by Markus Weninger *et al.*, 2020; permission conveyed through Copyright Clearance Center, Inc.

*Vasco* described how they use visualization to reuse some objects and reduce the number of allocations and garbage collections.

- *Memory fragmentation.* This issue related to a failure at reusing memory that has been released. Furthermore, excessive fragmentation over memory may lead to more costly performance behavior. We found that 13.04% of the visualizations help developers identify memory fragmentation. For instance, article **S31** [93] allows users to analyze the behavior of a memory allocator by displaying the memory accesses through time. This visualization enables developers to identify unnecessary fragmentation since free memory blocks can be detected quickly.

We also noticed that selected articles usually define the roles of users of visualizations. We detected that 13.04% of the visualizations help students or novice developers analyze memory consumption. For example, article **S3** [63] presents *PVC* to support students with understanding the program execution status and behavior. The authors of this article experimented with 35 university students to evaluate the usability of *PVC*. Furthermore, most of the visualizations (89.13%) assist developers and software engineers. Some of these articles determine that proposed visualizations are suitable for supporting developers with experience in software engineering and with knowledge of memory management.

**RQ1:** Software visualizations are designed to support programmers for different purposes. We detected five focus points of analysis: (i) memory usage analysis in specific architecture (23.91%), (ii) data structure analysis (30.46%), (iii) cache performance analysis (10.86%), (iv) memory regression (0.46%), and (v) general purposes such as memory access or heap usage analysis (32.60%). We also found that around 47.82% of software visualizations help to identify memory issues.

## 2.3.2 RQ2: Data

Monitoring and analyzing memory usage is a complex task for developers since it is necessary to collect and examine different software aspects. The selected articles usually present detailed sections to describe the data collection. We noticed that several articles implement a profiler to gather information. Other studies use dedicated tools for this purpose, such as *Pin* [80], *Jinsight* [39], *DynamoRIO*<sup>3</sup>, *etc.* Nonetheless, some studies only describe the information visualized, but do not explicitly mention how they collect the data.

We defined the classification scheme according to the sources from which various data were collected. As a result, the articles are categorized based on three sources: (i) program execution, (ii) source code, and (ii) version control systems. In this classification, software visualizations can belong to multiple categories, as shown in Table 2.11.

Table 2.11: Classification of articles based on the data source

Data source	Data	References	Total
Program execution	Memory allocations	S4, S7, S14, S19, S21, S23, S25-S27, S33, S39, S45	12
	Memory allocations and release	S1-S2, S12, S18, S28-S29, S31-S32, S36-S38, S40, S43	13
	Memory events	S3, S5-S6, S8-S11, S13, S15-S17, S20, S22, S24, S30, S34-S35, S41-S42, S44, S46	21
	Relationships between functions/methods	S1, S6-S7, S10-S11, S14, S21, S40	8
	Variable references	S1-S3, S12, S19, S23, S25-S29, S39-S40, S43, S45	15
	Time	S4-S6, S11, S16, S30-S31, S33, S36, S40-S41, S44	12
	Threads	S5, S6, S8, S10, S13, S15-S17, S24, S33, S36, S40	12
	Data shared between computational units	S5, S8-S9, S13, S15-S17, S35, S42, S46	10
Source code	Line of code	S3, S5-S6, S10, S17, S20, S22, S24, S29, S33, S45-S46	12
	Class	S1-S2, S7, S11, S14, S18, S21, S28, S36, S40	10
	Structural component	S1, S10-S11, S36	4
Version control system	Code changes	S4	1

**Program execution.** This category involves the articles that collect or calculate data from program execution. This information facilitates the understanding of the behavior of a program. All the articles extract various data from program execution to support developers with memory consumption analysis. However, we noticed that the information selected varies in different aspects.

During the program execution, a large number of memory events occur. Articles regularly mention that memory traces are collected. However, the concept of memory trace could be too general, so we focused on the details of the memory traces collected. We considered three memory events: (i) data is allocated in sections of memory (allocations), (ii) data allocated is

<sup>3</sup><http://dynamorio.org>

used (read and write), and (iii) the occupied memory that is not needed anymore should be released (deallocation).

- *Memory allocations.* We detected that 26.08% of the articles describe extracting data related to memory allocations but do not consider when the memory is released due to the difficulty of extracting this kind of information [116]. These articles usually provide visualizations that display the objects allocated during the program execution to identify objects that consume more memory and distinguish how these objects are related. However, according to our previous research [22], this information could be insufficient at helping developers detect optimization opportunities and address memory issues quickly.

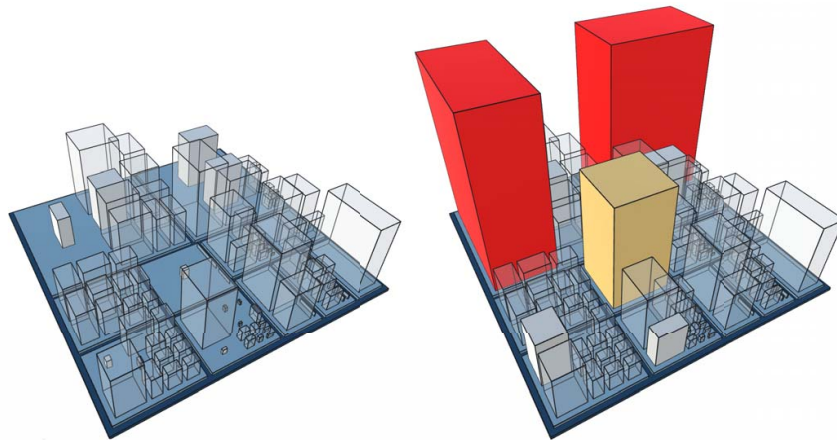


Figure 2.5: An example of *Memory cities* [157] representing the program’s heap shortly after startup (left) and 2 minutes and 300 garbage collections later (right). ©2020 Year IEEE. Reprinted, with permission, from “Memory Cities: Visualizing Heap Memory Evolution Using the Software City Metaphor” by Markus Weninger *et al.*, 2020.

- *Memory allocations and release.* We found that 28.26% of the articles determine gathering data from memory allocations and memory release by tracking specific instructions (*e.g.*, free, delete) or based on garbage collection events. For example, articles **S1** [156], and **S2** [157] abstract the heap memory evolution through time to assist developers in quickly detecting memory issues (*e.g.*, memory leaks, memory bloats). Figure 2.5 illustrates the approach proposed by article **S2**, which shows at the left an application’s heap visualized with memory cities shortly after startup and at the right the state of the heap 2 minutes and 300 garbage collections later.
- *Memory events.* A total of 45.65% articles collect data from all memory events described previously. Some of these articles present visualizations to support students or developers with understanding memory consumption. For example, article **S6** [27] proposes *Memoro*, a profiler with a visualization that shows how a program uses the memory. *Memoro* calculates useful metrics (lifetime, usage, useful lifetime) based on the data extracted (*e.g.*, number of reads, number of writes) from the memory accesses. These defined metrics allow developers to detect inefficient use of memory quickly. On the other hand,

other articles propose visualizations for specific aspects, such as cache performance analysis or memory analysis in HPC applications.

Furthermore, articles usually specify collecting metrics (*e.g.*, memory address, size) involved with each memory event. Additionally, some articles describe gathering additional information described below to assist developers with memory consumption analysis.

- *Relationships between functions/methods.* Extracting the calling relationships is a common strategy to assist developers with control-flow analysis [76]. Commercial tools (*e.g.*, JProfiler<sup>4</sup>, Yourkit<sup>5</sup>) display this information using a tree structure as shown in Figure 2.6.

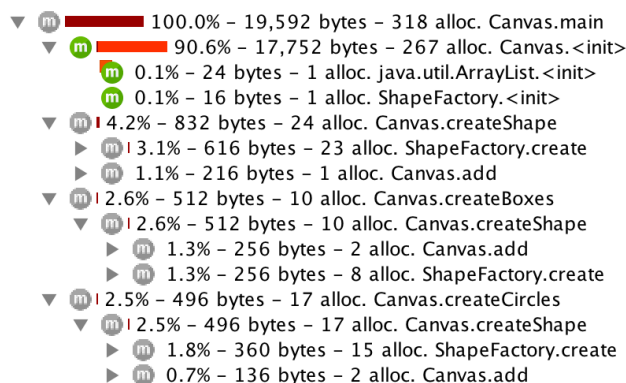


Figure 2.6: JProfiler displaying the methods executed with a Calling Context Tree.

We found that 17.39% of the articles describe collecting this information to determine how functions are related to memory events and track specific functions. To exemplify, article **S11** [54] helps developers understand memory consumption by extracting the memory accesses and the call stack. The visualization connects a dense scatter plot for memory accesses and a flame graph for the call stack. As a result, this visualization allows developers to explore through the memory accesses and determine which functions are involved.

- *References between variables.* Some articles proposed visualizations to support developers with data flow analysis. For this reason, 32.60% of the articles specify the extraction of references between variables. The articles focused on analyzing the memory consumption in object-oriented programming languages, which usually display the allocated objects and the references between these objects. For instance, article **S2** [157] proposes *Memory cities*, a visualization to inspect memory growth and reference patterns over objects. Weninger and colleagues employed *Memory cities* to identify memory leaks by examining reference patterns.
- *Time.* We found that 26.08% of the articles explicitly describe collecting how much time is spent executing some instruction or when a memory event occurs to facilitate the program understanding.

<sup>4</sup><https://www.ej-technologies.com/products/jprofiler/overview.html>

<sup>5</sup><https://www.yourkit.com>



- *Threads.* For 26.08% of the articles specify extracting which threads are involved in memory events. Article **S33** [116] describes that showing the threads created, destroyed and what each thread is doing is fundamental to show programmers detailed information about the program’s behavior.
- *Data shared between computational units.* We detect that 21.73% of the articles describe gathering information related to how memory resources are shared among processors. These articles present visualizations to assist developers in understanding the memory management between multiple processors.

**Source code.** The articles that present static aspects, which are inferred without executing the program belong to this category. We found that 47.82% of articles usually extract static information to help developers map data collected from program execution to source code. The latter benefits developers by identifying and proposing changes on the source code that reduce memory consumption or repair memory issues.

- *Line of code.* We detected that 26.08% of articles extract the file and line of code corresponding to memory events. These articles usually propose visualizations with interaction mechanisms to provide the line of code or a highlighted piece of source code for relating the data from program execution to source code quickly.

*Class.* We found that 21.74% of articles collect information at the level of class. As a result, developers can pinpoint classes with specific issues. For example, article **S18** [151] highlights classes that contain methods involved with several allocations or several deallocations.

- *Structural component.* We found that 8.69% of articles gather information about which package or module is involved with memory events. These articles usually present visualizations that group visual elements based on a structural component. For instance, article **S11** [54] displays the functions executed through visual elements and assigns the color of these visual elements based on the module as shown in Figure 2.7.

**Version control systems.** Only article **S4** [123] describes extracting data from commits or changes on source code between versions. This article proposes a graph of glyphs to identify memory regressions between consecutive commits.

**RQ2:** All software visualizations show data from the program execution. The most popular data from program execution involves memory events (allocations, accesses, releases) and references between variables. About 47.82% of software visualizations display static information. Only one visual approach displays information about changes between versions.

### 2.3.3 RQ3: Visual Representation

This section covers the analysis and categorization of the selected papers based on three relevant aspects in the software visualization field: visual techniques, interactions, and medium.

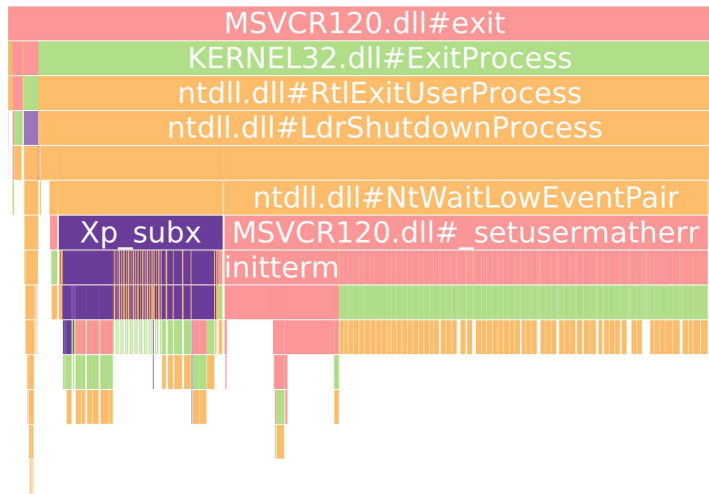


Figure 2.7: Gralka *et al.* [54] illustrate the application’s call stack over time as a flame graph. Each box corresponds to a method called in the application. In this case, colors are assigned based on modules. ©2017 Year IEEE. Reprinted, with permission, from “Visual Exploration of Memory Traces and Call Stacks” by Patrick Gralka *et al.*

Table 2.12: Classification of articles based on the visual technique

Visual techniques	References	Total
Geometrically-transformed	S1-S3, S5, S7, S10, S14, S16-S20, S22-S23, S25-S27, S29, S33, S39, S40, S43, S45	23
Iconic	S2, S4-S5, S7, S14, S22, S25, S28, S33, S36, S40, S43	12
Dense Pixel	S8, S11, S13, S15, S24, S30, S31-S32, S35, S37-S38, S41	12
Stacked	S2, S6, S9, S11, S12, S21, S28, S29, S33, S36, S40	11
Standard 2D/3D	S1, S5-S6, S9-S10, S14-S15, S34, S42, S44, S46	11

### 2.3.3.1 Visual techniques

Authors proposing a software visualization employ different visual techniques to explore the information collected from a software application. We categorize the articles according to the classification scheme proposed by Keim [66]. As a result, Table 2.12 illustrates the distribution of articles based on five categories. In the following, we describe the results of these categories.

- *Geometrically-transformed.* According to Keim [66] geometrically transformed techniques transform multidimensional data into low dimensional data. This transformation involves mapping an object to a set of points and lines in 2D or 3D (*e.g.*, node-link diagrams, parallel coordinates). Half of the selected articles (50%) employ geometrically transformed techniques. This category is the most frequent since several authors propose node-link diagrams to represent relationships, such as object references or the relationships between functions. For instance, article **S26** [9] proposes *Heapviz* to explore and identify primary data structures. *Heapviz* displays a node-link diagram where the nodes represent object instances, and the edges denote the references between



objects as shown in Figure 2.8. *Heapviz* allows developers to identify populated data structures, data structures containing other data structures, and objects referenced by several data structures.

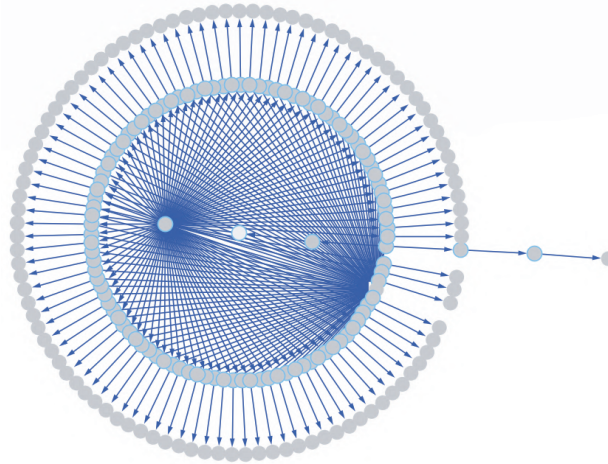


Figure 2.8: *Heapviz* [9] presents an interactive visualization to support the analysis of data structures. ©Republished with permission of ACM (Association for Computing Machinery), from “Heapviz: interactive heap visualization for program understanding and debugging”, by Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. 2010. Permission conveyed through Copyright Clearance Center, Inc.

- *Iconic displays.* This category involves visual techniques, which map the multidimensional data attributes to icon features (*e.g.*, tile bars, star icons). As a result, iconic techniques display icons whose characteristics vary concerning the data attributes. Of the selected articles, 26.08% employ iconic techniques. For example, **S4** [123] introduced the glyph called *Spark Circle* to analyze the variations of metrics (objects allocation variation, number of changed methods, execution time variation) between consecutive commits in a commit-graph visualization as shown in Figure 2.9. In this visualization, each spark circle has three segments, the pink segment for the number of changed methods, the orange segment for the objects allocation variation, and the blue segment for execution time variation. The height of each segment is proportional to the absolute value of the respective metric, and the border is black if any metric increases. As a result, the authors of this visualization detected performance and memory regressions.
- *Dense pixel.* This category includes techniques that represent data values as pixels and group them based on their dimension in specific areas (*e.g.*, matrix visualizations). In total, 26.08% of the selected articles use dense pixel techniques to represent a large amount of data (*e.g.*, memory accesses). For example, article **S31** [93] proposes a visualization tool shown in Figure 2.10 to analyze the behavior of memory allocators in C programs. The main view presents an orthogonal dense pixel layout of time versus memory addresses, which displays hundreds of thousands of allocation events without wasting screen space. The rectangular sizes represent the lifetime and size of blocks, and the color displays the allocation process. This visualization allows developers to analyze memory allocators to optimize their functionality for reducing fragmentation.

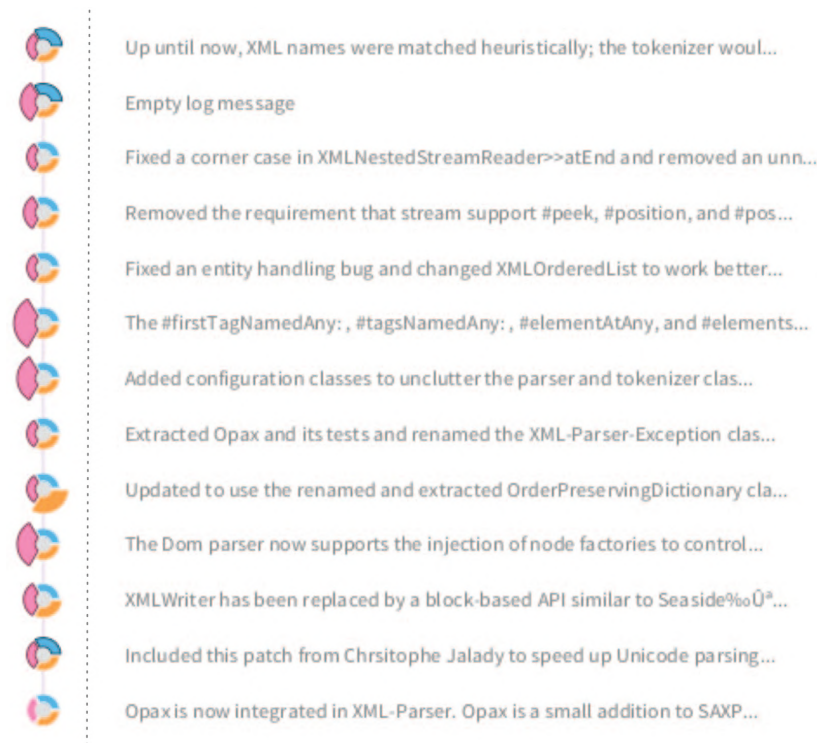


Figure 2.9: Visualization proposed by Sandoval [123] to analyze variations between commits. ©2019 Year IEEE. Reprinted, with permission, from “Enhancing Commit Graphs with Visual Runtime Clues” by Juan Pablo Sandoval Alcocer, 2019.

- *Stacked displays.* This category includes visual techniques that show data with a hierarchy structure (*e.g.*, treemaps [130], hierarchical stacking). Of the articles, 23.91% employ stacked displays to represent hierarchical partitioning. To illustrate, Figure 2.11 shows *Vasco*, an interactive visualization to explore object churn proposed in article **S21** [46]. *Vasco* represents the calling relationships between functions by employing a sunburst. *Vasco* allows users to detect problematical functions by mapping the color and angle to different metrics (*e.g.*, number of allocated objects, number of captured objects). As a result, users can explore functions that allocate many objects that are eventually released and which functions release them. The authors of *Vasco* demonstrated how to employ their visualization to find and solve memory churn.
- *Standard 2D/3D.* The articles which describe techniques such as plots of two or three dimensions (x-axis, y-axis, and z-axis) belong to this category. In total, 23.91% of the selected papers present standard 2D/3D displays (*e.g.*, bar charts, pie charts). For instance, article **S34** [111] employs standard techniques to analyze cache behavior. *YACO* support developer on understanding cache performance by displaying several bar charts and pie charts to present the statistics on cache hits and misses.

Finally, we found that 39.13% of the selected studies employ more than one visual technique. Consequently, the most popular combination of visual techniques involves the geometrically-transformed display with iconic display.

**Views.** All the visualizations display the information in one or more views. Commonly, the

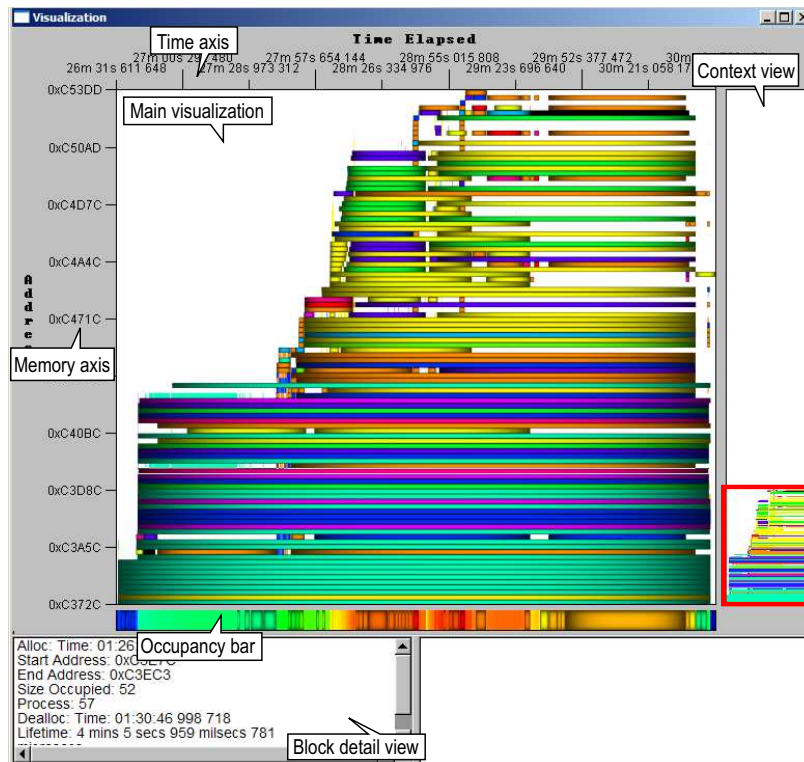


Figure 2.10: Visualization proposed by Moreta and Telea [93] to analyze memory allocations behavior. ©2007 Year IEEE. Reprinted, with permission, from “Visualizing Dynamic Memory Allocations” by Sergio Moreta, 2007.

use of well-integrated multiple views facilitates the exploration of distinct aspects of the data. To illustrate the number of views used on the selected papers, we reviewed the visualization description provided in each one. We found that 47.82% of the studies describe using a single view to display all the information. Most of these papers enrich their visualizations by combining two or more visual techniques, as we described previously. We also noticed that 39.13% of the articles report employing between two to four views. Finally, we detected that 13.04% of the articles adopt more than four views, usually to display other aspects through visualizations with standard 2D/3D techniques.

**RQ3.1:** Around 60.87% of software visualizations employ one visual technique, and the remaining use two or more techniques. The most popular technique (50%) is the geometrically-transformed.

### 2.3.3.2 Interactions

Some visualizations increased their effectiveness by providing interaction options to the users. Usually, a practitioner has the intention of performing some actions over the graphics to facilitate information analysis. However, few articles explicitly describe the supported interactions. This information is usually mixed with the visualization description or a section on usage scenarios. In a number of cases this information was placed in some footnotes. In

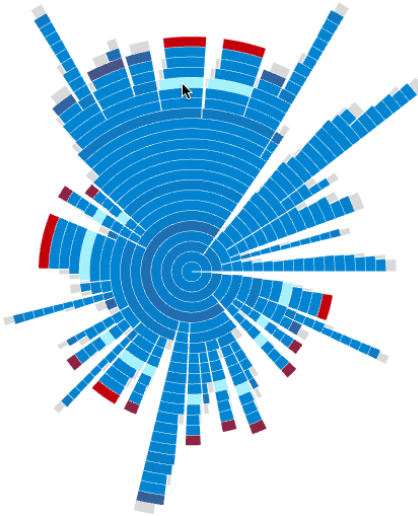


Figure 2.11: *Vasco* [46], an interactive visualization to explore object churn. ©2012 Year IEEE. Reprinted, with permission, from “Vasco: A visual approach to explore object churn in framework-intensive applications” by Fleur Duseau, 2012.

order to analyze the interaction options that proposed visualizations support, we resorted to classifying only articles that explicitly specify the supported interactions. This classification was based on the taxonomy proposed by Yi *et al.* [167]:

- *Select: mark something as interesting.* Distinguish visual elements of interest is relevant for dense visualizations. We found that 60.86% of the visualizations support this interaction. For example, article **S2** [157] presents *Memory cities*, a visualization to analyze heap evolution using the software city metaphor. In this visualization, the buildings are colored using a gradient ranging from gray to red. *Memory cities* allow the user to highlight a building in blue and thus facilitate its tracking over evolution.
- *Explore: show me something else.* A user can view a limited amount of graphic elements due to a large amount of data and the screen space used to display them. Users usually are interested in seeking out something new by moving the camera across a scene. This category includes interaction techniques (*e.g.*, panning) that allow users to explore different sub-collections of data. We observed that 54.34% of the articles provided exploration techniques. For instance, article **S2** [157] allows moving the camera to view the visualization from above, with a perspective as though the observer were a bird for facilitating inspection of visual elements.
- *Reconfigure: show me a different arrangement.* The arrangement of elements on the screen helps analyze data. We noticed that 17.39% of the visualizations support this task, like article **S7** [22] presents a node-link diagram that allows users to modify the layout by dragging nodes.
- *Encode: show me a different representation.* This category involves the interactions that enable a user to modify the visual representation. We found that 10.86% of the visualizations support metric selection like article **S21** [46] that presents *Vasco* that provides a menu bar to change the metrics for color or size of arcs.

- *Abstract/Elaborate: show me more or less detail.* To examine the details of an element of interest is a primary task. Therefore, this category includes details-on-demand interactions. According to Yi and colleagues, the interactions in this category allow developers to adjust the level of abstraction of a data representation. This category is the most frequent in visualizations (60.86%). Usually, the visualizations provide pop-up windows or provide panels with detailed information.
- *Filter: show me something conditionally.* Filtering according to criteria allows users to focus on specific elements quickly. We detected that 32.60% of visualizations enable users to hide elements that do not satisfy a condition. For example, article **S7** [22] presents a menu for excluding methods based on the type of objects that they allocate.
- *Connect: show me related items.* Users focusing on an element of interest will typically explore its relationships with other elements. We found that 21.74% of the visualizations provide interactions to support the navigation through the related elements. For instance, article **S7** [22] facilitate this task by highlighting the edges and nodes related to a selected node.

Additionally, we found that 28.26% of the articles do not explicitly describe the interaction mechanisms provided or specify the intentions of users when they interact with visualizations. We also noticed that the visualization mantra “Overview first, zoom and filter, then details on demand” [131] is not always considered by the proposed visualizations.

**RQ3.2:** Around 71.74% of software visualizations provide interaction mechanisms. The most popular interactions involve (i) distinguishing elements of interest (select) and (ii) examining the details of an element of interest (abstract/elaborate).

### 2.3.3.3 Medium

Advanced technology provides users different ways to interact with 3D or 2D visualizations. Maletic and colleagues [81] explained that mediums (*e.g.*, single monitor, wall displays, immersive virtual reality environments) might improve visual representations since they present distinct characteristics. Although monitoring resource consumption may involve some dedicated devices [86], most of the selected visualizations are rendered on a standard monitor of a desktop computer or laptop. Some articles do not explicitly provide the medium, but we inferred that visualizations are rendered on a standard computer screen. As a result, we found that no study exploited the medium to enhance visualizations dedicated to supporting memory consumption analysis tasks.

**RQ3.3:** Most software visualizations are rendered on a standard monitor of a desktop computer or laptop. Some articles do not explicitly mention the medium used.

## 2.3.4 RQ4: Evaluation

This section describes the distinct evaluation strategies to validate the effectiveness of the selected software visualizations. We classify the selected studies in three categories based on the work of Merino *et al.* [84]: theoretical, empirical and no explicit evaluation. Table 2.13 illustrates the distribution of articles based on the strategies used to evaluate visualizations.

Table 2.13: Classification of articles based on the strategies used to evaluate visualizations

Categories	Strategy	References	Total
Empirical	Usage scenario	S2, S4-S6, S8-S13, S15, S17, S19-S23, S25-S27, S29-S31, S35, S38, S41	26
	Anecdotal evidence	S14, S19, S24, S28, S43	5
	Experiment	S1, S3, S7	3
No explicit evaluation		S16, S18, S32-S34, S36-S37, S39-S40, S42, S44-S46	13

We found that 28.26% of the articles do not provide an evaluation, while the remaining present an empirical evaluation. These empirical evaluations are divided into subcategories described below.

- *Usage scenario.* Of the selected studies, 56.52% only provide application examples. These usage scenarios provide an extended description of how to address memory issues or analyze memory usage with the proposed software visualization. The authors highlight the interactions and the advantages of their visualizations by analyzing popular benchmarks like *DaCapo suite* [21], *DB suite*, *Reptile* [153], *GCOld* [107], *Paraffins*, or open-source projects. Half of the papers in this category presented usage scenarios as case studies. Nonetheless, they do not explicitly describe that professional developers in the industry context with real-world applications employ the visualization. Most of the authors usually give an extended description to demonstrate the effectiveness of their visualization in different cases. However, this description is limited to providing the article authors’ experience in using their tool, bearing the risk of biased conclusions according to different articles [162, 169].
- *Anecdotal evidence.* We found that four articles present a short section, usually with the title “industrial experience”, where they informally describe the use of the visualization on software companies with professional engineers. In this way, they claim the effectiveness of the visualization, but they do not present data of formal interviews or questionnaires. For example, article **S24** [118] collects information from an informal interview to four programmers with the think-aloud method. The goal of this interview is to collect information about how developers employ *Allocray* to detect memory leaks. This research summarizes the usability observations and the feedback of the participants during the interview.
- *Experiment.* Three articles present experiments with participants over software applications. For instance, paper **S7** [22] carries out a user study to evaluate their visualization.

The authors explain with details the interviews with eight participants, who use the tool to achieve some tasks. This study describes the results and observations during the work sessions and gathered feedback from the participants.

Finally, none of the selected articles evaluate their visualizations with professional developers and real-world software applications in the context of the software industry. As we mentioned in the category of use scenarios, 28.26% of the articles present sections titled “Case studies”, however the users involved in the evaluation are the authors of the articles. According to Merino and colleagues [84], a study that provides an evaluation with authors instead of independent developers, is considered a “Usage Scenario”.

**RQ4:** Most software visualizations are evaluated empirically. About 56.52% of software visualizations provide usage scenarios as an evaluation strategy. However, most articles lack robust empirical evaluation of how visualizations perform in practice with software developers and real-world applications.

### 2.3.5 RQ5: Availability

This section lists the selected software visualizations that are available. Most of the software visualizations support a group of people to address a problem of a particular context, this group of people is commonly called *audience* [81]. Having an available tool benefits the target audience to perform software engineering tasks over real-world applications. Also, researchers can conduct how their approach works compared with other visualization proposals by using controlled experiments.

We extracted from the selected articles the tool’s name and the links from which the visualization tool is available. However, only 21.73% of the articles provide an existing link, and 6.52% of the articles present a non-existing link. Additionally, we performed web searches based on the article’s title, the authors, and the tool’s name to find visualization tools available for the remaining articles. As a result, we identified links to visualization tools for 23.91% of the articles due to web search. Table 2.14 details the information of available software visualizations tools. Table 2.14 presents the tool’s name, where the link was found (article content or web search), the link, and extra information presented aside from the article to install and use the software visualization.

**Extra information.** We focused on the variety of information that could be available to enrich the experience and facilitate the use of the visualization. Therefore, we detected the presence of the following information:

- *Video.* Some links provide a video explaining features of the visualization. In this case, five links present a video showing the use of the visualization as supplemental material.
- *Sample data.* Some conferences promote the release of datasets to gain public data and the possibility of replicability. Six links provide a list of sample data, which reference the data collected from the applications used in the article as examples.



Table 2.14: Visualization tools and additional information from the selected articles. The information was verified on 18/05/2021.

ID	Ref.	Tool	Location	Link	Video	Sample Data
S1	[156]	AntTracks	Article	<a href="http://mevss.jku.at/?page_id=1592">http://mevss.jku.at/?page_id=1592</a>		
S2	[157]	Memory Cities	Article	<a href="https://doi.org/10.5281/zenodo.3991785">https://doi.org/10.5281/zenodo.3991785</a>	✓	✓
S3	[63]	PVC	Article	<a href="https://github.com/RYOSKATE/PlayVisualizerC.js">https://github.com/RYOSKATE/PlayVisualizerC.js</a>		✓
S6	[27]	Memoro	Article	<a href="https://github.com/epfl-vlsc/memoro">https://github.com/epfl-vlsc/memoro</a>		
S7	[22]	–	Article	<a href="http://dx.doi.org/10.5281/zenodo.1311787">http://dx.doi.org/10.5281/zenodo.1311787</a>	✓	
S8	[145]	NumaMMA	Article	<a href="https://github.com/numamma/numamma">https://github.com/numamma/numamma</a>		✓
S9	[53]	MemAxes	Web search	<a href="https://github.com/LLNL/MemAxes">https://github.com/LLNL/MemAxes</a>		✓
S13	[44]	Aftermath	Web search	<a href="https://www.aftermath-tracing.com/installation/">https://www.aftermath-tracing.com/installation/</a>	✓	
S14	[62]	Memory blueprint	Web search	<a href="http://smalltalkhub.com/ainfante/MemoryProfiler/">http://smalltalkhub.com/ainfante/MemoryProfiler/</a>		
S15	[18]	Tabarnac	Article	<a href="https://github.com/dbeniamine/Tabarnac">https://github.com/dbeniamine/Tabarnac</a>		
S19	[82]	HeapDbg	Article	<a href="http://heapdbg.codeplex.com">http://heapdbg.codeplex.com</a>		
S21	[46]	Vasco	Web search	<a href="http://geodes.iro.umontreal.ca/en/projects/vasco/">http://geodes.iro.umontreal.ca/en/projects/vasco/</a>		✓
S26	[9]	Heapviz	Web search	<a href="https://github.com/eaftan/heapviz">https://github.com/eaftan/heapviz</a>		
S28	[115]	Dyvisе	Article	<a href="ftp://ftp.cs.brown.edu/u/spr/dyvisе.tar.gz">ftp://ftp.cs.brown.edu/u/spr/dyvisе.tar.gz</a>		
S31	[93]	–	Web search	<a href="http://www.staff.science.uu.nl/~telea001/uploads/Software/MemoView/">http://www.staff.science.uu.nl/~telea001/uploads/Software/MemoView/</a>		
S33	[116]	Jove	Web search	<a href="http://cs.brown.edu/~spr/research/visjove.html">http://cs.brown.edu/~spr/research/visjove.html</a>	✓	
S36	[114]	Jive	Web search	<a href="http://cs.brown.edu/~spr/research/vizjive.html">http://cs.brown.edu/~spr/research/vizjive.html</a>	✓	
S37	[109]	GCspy	Web search	<a href="https://www.cs.kent.ac.uk/projects/gc/gcspy/">https://www.cs.kent.ac.uk/projects/gc/gcspy/</a>		
S39	[171]	Memory graphs	Article	<a href="http://www.st.cs.uni-sb.de/memgraphs/">http://www.st.cs.uni-sb.de/memgraphs/</a>		✓
S41	[168]	–	Web search	<a href="http://www.cs.toronto.edu/~yijun/cacheviz.guide.html">http://www.cs.toronto.edu/~yijun/cacheviz.guide.html</a>		
S45	[170]	DDD	Web search	<a href="https://www.gnu.org/software/ddd/">https://www.gnu.org/software/ddd/</a>		

**RQ5:** Overall, around 54.36% of the visualizations are not available. Furthermore, only 21.37% of software visualizations are available through a valid link in the articles, and about 23.91% of visualizations are available on the web.



## 2.4 Discussion and Open Challenges

The results described previously provide a general overview of the state-of-art software visualizations centered on the analysis of memory consumption. We have described distinct features and categorized the selected software visualizations to answer our research questions. This section discusses some findings, open challenges for our proposed dimensions, and highlights some initial observations. Also, we provide recommendations to practitioners and researchers based on our research questions.

**Tasks supported.** The selected software visualizations attempt to facilitate the analysis and solution of several memory issues (*e.g.*, memory bloat or fragmentation). The design of these visualizations is based on assumptions of developers' needs about the memory issue to be addressed. However, to our knowledge, developer design requirements or needs for each particular memory issue has not been thoroughly researched yet. In the past, several studies have analyzed what developers asked during software development [76, 133], revealing a number of needs to be addressed. However, no study provides detailed questions related to memory consumption analysis. Having solid knowledge about developers' needs while addressing these issues may help improve the design and effectiveness of the proposed visualization tools.

**Observation 1:** Programmers' needs during memory usage analysis and memory issue repairing are not entirely studied.

Section 2.3.1 details the classification of articles based on the provided tasks to support programmers over the analysis of memory usage. According to our findings, various visualizations help developers with memory consumption analysis by focusing on different aspects. More than half of the visualizations in *General* and *Data structure* are available. However, few visualizations are available to assist developers in analyzing applications with specific architectures, and one visualization is available for analyzing cache behavior. The unique visualization to analyze memory regression is not available. We also detected that at least two visualizations are available to detect each type of memory issue. However, the number of visualizations available is reduced.

**Observation 2:** Domain-specific memory analysis, memory issue identification, and memory regression analysis are not fully explored yet, leaving an open opportunity.

**Data abstracted.** Section 2.3.2 describes the aspects of the software involved in the analysis of memory consumption. According to our findings, a set of articles develop a strategy to gather information, while others use dedicated tools, and the data extracted by these tools are from different projects. The variety of analyzed projects, tools, and data collection strategies makes it difficult to compare proposed visualizations. However, creating a baseline of project set (*i.e.*, projects with particular memory issues) and collection strategies may offer developers and future researchers a guide to successfully gathering specific data and baselines to contrast their tools with state of the art.

**Observation 3:** Comparing proposed visualizations is difficult due to the variety of data collection strategies, analyzed projects, and the availability of tools.

Regarding the aspects extracted, we found that most visualizations dismiss mapping the information from program execution with information from source code, like lines of code or classes. Consequently, developers may deal with problems detecting which part of the code is causing or participating in a memory issue. Relating memory metrics with source code is still an open area for further research.

**Observation 4:** Connecting dynamic information to static information is not popular in software visualizations for this context. However, this could facilitate detecting the causes of memory anomalies.

**Visual representation.** Section 2.3.3 details the visual techniques used, the interaction options supported, and the medium where the visualization is displayed. We found specific trends in visualizations when using some visual techniques depending on the domain of the problem. For example, most visualizations that assist developers with data structure analysis employ geometrically transformed techniques. However, there is no evidence of the advantages of using a particular visual representation for a single problem domain. In addition, we found that most of the articles present multiple views to display the information. In the same way, we do not observe if using a single view presents better, similar, or worse results than using multiple views.

**Observation 5:** Some software visualizations present different techniques to represent the same data. However, it is not known which technique is more suitable.

Finally, we detected that most of the studies employ a single monitor screen to render the visualization. We encourage researchers to analyze the impact of the medium on the effectiveness of visualizations centered on support memory consumption analysis by employing different mediums to render the visualization, like wall-display, multi-touch tables, or a 3D immersive environment.

**Observation 6:** The use of other medium (*e.g.*, 3D immerse environment, wall-display) than single monitor screen is not explored for software visualizations focused in support memory usage analysis.

**Evaluation.** Section 2.3.4 summarizes the evaluation strategies used by the selected articles. We found that most articles lack robust empirical evaluation that involves software developers. For instance, we detected that only three articles present evaluations with users of the target audience and expose the comments and observations during the work sessions. We also observed usage scenarios presented as case studies, which detail the author's experience in employing the proposed visualization to analyze memory consumption.

**Observation 7:** There is a limited empirical evidence about how programmers employ software visualizations when analyzing memory usage and addressing memory issues.

Conducting experiments could be difficult because the nature of the problem domain may require expert developers with a high level of knowledge regarding memory management. Besides, designing and conducting robust experiments is an aspect that memory visualization articles need to improve.

**Availability.** Actually, only 21.73% of the articles present a valid link where the software visualization tool is available. We detected three articles published between 2002 and 1997 that provide no valid links. Additionally, we found links with software visualization tools for 23.91% of the articles by performing a search on the web that could be tedious, as we explain in Section 2.3.5. We must highlight that we did not try to install the tool nor verify whether it works. However, we enlisted the additional information (videos that show how to use it correctly or a data sample) that the link presents to support users with the visualization tool.

**Observation 8:** Programmers may fail to adopt software visualizations to perform tasks related to memory consumption analysis and memory issues detection due to the lack of visualizations available for their use.

## 2.5 Related Work

To the best of our knowledge, this work is the first literature review of software visualizations focused on supporting the user to comprehend memory consumption. Nevertheless, relevant work was published in the software visualization field covering different aspects over the years [105, 106, 119].

**Scope.** Focus on software visualizations over a general context: these surveys [13, 105, 106, 119, 143], systematic literature reviews [68, 83, 85, 95, 142, 149], taxonomies or classifications [43, 81] generate findings of how visualizations support users on software engineering tasks. In addition, a number of studies cover visualizations supporting specific aspects of the software engineering field. For example, there are literature reviews [67, 129, 158] focused on software visualizations to support the analysis of software architecture design. These studies analyze how software architecture is visually represented to examine the design based on features like complexity, cohesion, *etc.* Another popular domain is software evolution. For this domain, Novais *et al.* [101], and Salameh *et al.* [121] published systematic studies centered on visualizations to display how certain software elements (*e.g.*, source code, dependencies) change over time.

In addition, several studies [10, 58, 64] focus on using software visualization in educational programming. These studies examine the benefits of using visualizations to improve and facilitate students' learning process. The studies consider the effectiveness of software visualization in engaging students in the field of education. Our study detected six articles

that propose software visualizations to help students or novice developers analyze memory consumption.

Furthermore, Bedu *et al.* [17] presented a tertiary systematic literature review on software visualization. This article identifies topic (*e.g.*, architecture, education) trends of surveys focused on software visualizations and issues related to software visualizations (*e.g.*, scalability, validation).

**Dimensions.** Furthermore, most of the surveys and systematic literature reviews [81, 83, 85, 106, 105, 119] cover the tasks that are supported, the gathered information, and the visual techniques used to display the information. The main variation is our survey's scope; consequently, the tasks supported and the collected information are more specific than in prior works. For example, we discussed that our visualizations under study help developers analyze the program behavior and support debugging tasks following the classification scheme of prior work. However, our findings show various focus points (*e.g.*, data structures, cache behavior) to analyze and different memory issues (*e.g.*, memory leak, memory bloat) to address. We also provided which data (*e.g.*, threads, time) and which information sources (*e.g.*, program execution, source code) are collected, similar to the study of Merino *et al.* [85]. However, we considered how the extracted data from different sources is related to helping developers with memory consumption analysis.

Furthermore, a minor number of the studies mentioned in this section cover the evaluation and availability dimension. However, some systematic reviews focus explicitly on how software visualizations are evaluated. The cases by Merino *et al.* [84], Sensalire *et al.* [127], and Seriai *et al.* [128] examine the different evaluation strategies to validate certain features of a software visualization study (*e.g.*, effectiveness, usability). These studies provide guidelines to produce enough evidence to evaluate software visualizations and describe some challenges in the field. They also explain the weak empirical evidence among software visualizations and detail the inconsistencies in the studies. Our findings expose that 73.91% of the articles present empirical evaluations, mostly usage scenarios. However, the number of studies that describe experiments and case studies is minor. Our results confirm that few articles evaluate visualizations with developers and real-world applications as prior work details.

We noticed that few studies [30, 85] examine the availability of software visualizations. However, our study does not limit publications' data and focuses on visualizations that support memory consumption. Consequently, we provide links to visualization tools not considered by the prior work.

**Methodology.** As mentioned, most of the relevant prior work focuses on reviewing the state-of-art in the software visualization field. These studies also follow the steps proposed in distinct guidelines for systematic reviews [70, 104]. However, they present differences with our work in some steps. For example, the construction of the search string could be less complex due to the scope of the studies. Therefore, the number of articles resulting from searching over digital databases and the number of selected papers tend to be higher than ours.

## 2.6 Threats to validity

Our study and results are subject to validity threats. To carefully identify possible threats and analyze how their impact may be mitigated, we decided the following:

***Search of articles.*** A threat to the validity of this study may be not cover all the relevant articles. We performed a systematic search to find articles that propose visualization centered on supporting developers with memory consumption analysis. We developed our search query based on keywords from articles that belong to our scope published between 2017 and 2020 in the most cited venues dedicated to software visualization or memory management. However, our search query is biased by the specific keywords of this set of articles. We decided to decrease this threat by performing an additional manual search and bi-directional snowballing. These two phases assisted in our finding of missing relevant studies.

***Selection of articles.*** A relevant article may be excluded during the selection phase and vice versa. We defined inclusion/exclusion criteria and a quality assessment to reduce bias in selecting articles. During the selection of inclusion/exclusion criteria, we independently review the title and the abstract to consider if an article should be included or not. We calculated the Fleiss' kappa for the inter-rater reliability, and the result was 0,72%, which is generally considered a good agreement. We then discussed and resolved the disagreements during meetings. For the quality assessment, we adopted a checklist to examine the quality of papers. The discrepancies found were reviewed again in a second iteration, and discussion sessions were carried out to reach a consensus.

***Data extraction.*** Another threat to consider is that the data extraction process could be biased. We mitigated this threat by establishing a protocol to extract the data for each paper. The thesis author managed a spreadsheet to keep records of relevant text segments and identify irregularities like missing information. The two thesis supervisors review if the data extracted was correct.

***Data analysis.*** During the data analysis, we performed thematic analysis and content analysis to answer our research questions. The thesis author performed a systematic process to conduct a thematic analysis for RQ1 and RQ2. This process includes generating codes and defining themes (patterns) that help answer the research questions. The codes and themes generated vary depending on the coder's experience, point of view, and level of abstraction. For example, to respond to RQ1, we detected visualizations focused on analyzing specific points. However, some articles were too general at determining their objectives, so we decided to consider these articles as a general-purpose group since no specific pattern was found. We tried to reduce this threat by checking the consistency of the process. Due to this, the two thesis supervisors examined the description of themes and the data coding. We carried out three discussion meetings to analyze the codes and the themes generated. As a result, we solve the differences.

To answer RQ3.1, RQ3.2, and RQ4, we conducted a content analysis. We selected classification schemes proposed in previous studies. We code the data based on these schemes and measure the agreement between the thesis author and the thesis supervisors. We detected

some specific discrepancies that were discussed and solved during a meeting.

## 2.7 Summary

This chapter summarizes the software visualizations that support programmers in analyzing memory consumption or addressing memory issues. We systematically selected 46 articles and categorized these articles based on five dimensions (i) the tasks supported, (ii) data abstracted, (iii) visual representation employed (visual techniques, interactions mechanisms provided, and the medium used), (iv) evaluations conducted, and (v) tools availability. The following paragraphs summarize our results about the five dimensions.

**Tasks supported.** We use thematic analysis to generate the categories based on the information about the purpose of the visualization to help the programmer in some tasks related to memory consumption analysis. Therefore, we classified the articles based on the focus point analysis and memory issue detection. According to the classification in focus point analysis, most software visualizations focus on supporting programmers in data structure and general purposes involved with memory usage analysis, such as heap analysis. On the other hand, there is not much research on visualizations to support memory regression or cache performance analysis. We also noticed that around 47.82% of the visualizations support memory issue detection. Consequently, memory issue identification is not fully explored yet, leaving an open opportunity.

Initially, we focus on looking for an empirical catalog of the developers' requirements or needs during memory usage analysis or memory issues detection to use it as a classification scheme to categorize the articles. However, we noticed that no study provides detailed questions or information about the needs of programmers when analyzing memory consumption. We consider that providing a solid knowledge about developers' needs while addressing these issues may help improve the design and effectiveness of the proposed visualization tools.

**Data abstracted.** We defined the classification scheme according to the sources from which various data were collected. We found that all software visualizations show data from the program execution. The most popular data extracted from program execution involves memory events (allocations, accesses, releases) and references between variables. About 47.82% of software visualizations display static information. Only one visual approach displays information about changes between versions. Additionally, our findings show that most visualizations dismiss connecting the information from program execution with information from source code, like lines of code or classes. We consider that collecting information from both sources reduces the effort of practitioners to analyze memory consumption.

**Visual representation.** We categorize the articles according to the classification scheme proposed by Keim [66]. We found that 60.87% of software visualizations employ one visual technique. The most popular technique is the geometrically-transformed display, frequently used in articles that propose node-link diagrams representing relationships between elements. We also noticed that most software visualizations provide interaction mechanisms. However, the visualization mantra "Overview first, zoom and filter, then details on demand" [131] is

not always considered by the existing visualizations. Finally, most software visualizations are displayed using a standard monitor. We consider that visualizations could be implemented to use other mediums, such as tactile devices or 3D environments.

***Evaluation.*** We classify the selected studies based on the work of Merino *et al.* [84]. Our results show that most software visualizations are evaluated empirically. About 56.52% of software visualizations provide usage scenarios as an evaluation strategy and used popular benchmarks like *DaCapo suite* [21], *DB suite*, *Reptile* [153], *GCOld* [107], *Paraffins*, or open-source projects. However, most articles lack robust empirical evaluation of how visualizations perform in practice with software developers and real-world applications.

***Availability.*** We extracted from the selected articles the tool's name and the links from which the visualization tool is available. Only 21.73% of the articles provide an existing link. Additionally, we identified links to visualization tools for 23.91% of the articles due to web search. Overall, around 54.36% of the visualizations are not available. Consequently, we consider the lack of availability one of the main weak points in the field.

In this chapter, we found the popular features among current software visualizations, opportunities for improvement, and open challenges in the field. In the following chapter, we will focus on exploring some of the following points: (i) the usefulness of displaying aspects from software related with how the program runs and connects the source code with dynamic information, (ii) how a visualization performs in practice, and (iii) the programmers' needs during memory usage analysis.

# Chapter 3

## Visualizing Memory Consumption with Vismep

In this chapter, we introduce Vismep, an interactive visualization prototype to assist programmers in analyzing the memory usage of Python programs. Vismep summarizes how the program runs and allocates memory using polymetric views [74]. Vismep also connects the source code with dynamic information. Chapter 2 positions the proposed visualization against the state of the art, where the main differences are the connection between source code and dynamic information, as well as displaying metrics focused on how the programs run (*e.g.*, calling relationship between functions/methods). Therefore, we explored how valuable this information is for practitioners. Consequently, this chapter also presents an exploratory study to understand how Vismep supports eleven programmers in practice and the perceptions of programmers about the tool. 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 its valuable views and high overall usability.

The content of this chapter is based on the publication “Visualizing Memory Consumption with Vismep” [24] (co-authored with Alexandre Bergel, Juan Pablo Sandoval, and Araceli Queirolo Cordova) and has been reformatted according to departmental guidelines. Additionally, this chapter is also related to the bachelor’s dissertation of Araceli Queirolo Cordova titled “Mejorar la usabilidad y efectividad de una herramienta de perfilamiento de memoria” [112]. The bachelor’s dissertation was supervised by the first supervisor and the author of this Ph.D. dissertation.

### 3.1 Introduction

Monitoring and understanding an application’s memory usage help programmers discover anomalies that may be non-trivial and could provoke crashes and performance degradation [52, 90, 139]. Numerous memory profiling tools have been proposed to assist programmers in monitoring memory usage and identifying memory anomalies [80, 99, 154]. 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 examine and address memory issues [22, 27, 54, 157] since visualizations are known to support practitioners in software comprehension [45, 74].

Although various tools are provided, some studies claim that programmers usually need substantial time to understand memory usage, detect anomalies, and repair issues [78, 166]. This situation could occur because little is known about what information programmers need when analyzing the memory usage of programs. In this chapter, we introduce a visual approach based on the state of the art and provide detailed information about the performance of this approach in practice. Consequently, we present a contribution in two relevant points mentioned in Chapter 2. Firstly, this chapter provides initial information about programmers' needs when analyzing memory usage. Secondly, we provide empirical evidence about how programmers employ our visual approach to find the information required to satisfy their needs. Obtaining this information represents a stepping stone in (i) enhancing the evaluations of software visualizations by providing detailed empirical evidence and (ii) an initial knowledge about the programmers' needs when analyzing memory usage. Consequently, our study could be valuable to demonstrate whether a tool provides adequate support, benefits, limitations, and how to improve the design.

As mentioned previously, this chapter 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 [74]. 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 chapter makes the following contributions:

- Vismep, an interactive visualization prototype that supports programmers in analyzing

memory consumption over Python applications. Vismep is publicly available<sup>1</sup>. We also presented an artifact that contains all the documentation necessary to install and execute Vismep in three different operative systems [48]. This artifact was accepted and received the “*Open Research Objects (ORO)*” and “*Reviewed Objects of Research (ROR)*” badges at the *ROSE Festival of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME 2022)*.

- 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.

**Structure of the chapter.** Section 3.2 summarizes the prior work. In Section 3.3, we describe in detail Vismep. Section 3.4 details the exploratory study conducted to understand how Vismep support programmers and how they perceive Vismep. Section 3.5 describes the results obtained about the information needs, Vismep usage and perception of Vismep. In Section 3.6, we discuss our findings and the limitations of the study. Section 3.7 provides the threats to validity of the study. Finally, Section 3.8 concludes and outlines future work.

## 3.2 Related Work

This section highlights and summarizes the background and literature related to (i) memory consumption analysis in Python, (ii) software visualizations for memory usage analysis, and (iii) the evaluation of those software visualizations.

### 3.2.1 Memory Consumption Analysis in Python

Several libraries and tools center on memory usage analysis in Python. Table 3.1 illustrates the tools/libraries along with the (i) activities they claim to support, (ii) information collected, and (iii) report presentation used. We extracted this information and other data (*e.g.*, installation requirements, links) from their respective documentation<sup>2</sup>.

These libraries and tools extract diverse information and report it using textual reports, non-interactive visualizations, and interactive visualizations. The libraries mentioned are highly expressive, flexible, and can generate tuned reports (primarily textual), but users must modify their code using the correct API calls. For instance, *memory\_profiler* [3] provides a decorator (`@profile`) to mark the functions to be profiled and report the memory used by each code line from the selected function. On the other hand, when practitioners employ some tools [8, 5] to extract information and show it usually through visualizations, they often select

---

<sup>1</sup><https://github.com/Balison/Vismep>

<sup>2</sup><https://www.dropbox.com/s/49mdqg5n11bhvdd/PythonMemoryProfilers.csv?dl=0>

Table 3.1: Libraries and tools along with (i) the activities that claim to support (A1 = Analyzing memory usage of entities; A2 = Analyzing allocation hotspots; A3 = Analyzing memory usage over time; A4 = Analyzing leaking objects), (ii) the information reported (M.A. = Memory allocations; M.R = Memory releases; R.F. = Relationships between functions; V.R. = Variable references; T. = Time; TH. = Threads; L.C. = Lines of code; C. = Class; S.C. = Structural component) and (iii) the report presentation used. The information was verified at 02/09/2022.

Library/Tool	Activities				Information reported									Report presentation	
					Program execution					Source code					
	A1	A2	A3	A4	M.A.	M.R.	R.F.	V.R.	T.	TH.	L.C.	C.	S.C.	Textual	Visualization
Guppy [1]	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
Muppy [2]	✓	✗	✓	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✓	✗
Objgraph [6]	✓	✗	✗	✓	✓	✗	✗	✓	✗	✗	✗	✓	✗	✗	✓
Memory_profiler [3]	✓	✓	✓	✗	✓	✓	✗	✗	✓	✗	✓	✗	✗	✓	✓
Tracemalloc [7]	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗	✓	✗	✓	✓	✗
Fil [5]	✓	✓	✗	✗	✓	✗	✓	✗	✗	✗	✓	✗	✓	✗	✓
vprof [8]	✓	✓	✓	✗	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓
Scalene [20]	✓	✓	✓	✗	✓	✓	✗	✗	✓	✓	✓	✗	✗	✓	✓
memray [4]	✓	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓

the data gathered using flags presented in the documentation without manually changing the code.

Both libraries and tools commonly claim to help programmers in the following activities:

**Analyzing memory usage of entities.** Some libraries and tools show the memory used by a particular entity (variable, function). For example, *Muppy*, *Guppy*, and *vprof* enlist the memory used by the allocations made during program execution. Other options [3, 4, 5, 7, 20] display the functions executed, and their memory consumed.

**Analyzing allocation hotspots.** Several libraries and tools report the code that allocates most memory (allocation hotspots). For instance, *Tracemalloc* and *memory\_profiler* enlists the allocation hotspots (code lines). *Fil*, *memray* highlight the code (line of code, function) responsible for allocating most memory using visual hints (*e.g.*, color, size).

**Analyzing memory usage over time.** Some libraries and tools [3, 4, 8, 20] display the memory usage over time through line charts or sparklines. Although other memory profilers [1, 2, 7] do not collect any time metric, they help users note changes in memory usage between points in time. For example, *Tracemalloc* shows how memory usage changes (increased, decreased) before and after executing a function.

**Analyzing leaking objects.** Few libraries [1, 2, 6] show the memory allocations made at a given point in time (after a function call) along with the objects that reference these allocations. For instance, *objgraph* displays a graph where each node is an object allocated in memory, and the edges represent the references between objects. These libraries allow programmers to detect memory leaks by locating unused memory resources.

This study not only introduces Vismep, but provides a detailed investigation of how programmers use Vismep to analyze Python applications. Also, we should mention that although Vismep collects information typically extracted by some of these libraries/tools, it

displays and connects the information differently. More specifically, we display four interactive views that employ different visual techniques (*e.g.*, geometrically-transformed displays, 2D displays) to summarize how the program runs and allocates memory instead of tables, textual reports, or non-interactive visualizations. Vismep also allows users to inspect the source code of the functions executed, something that most options do not consider or have limited support (*e.g.*, Tracemalloc shows only the lines that occupy the most memory).

### 3.2.2 Studies on Software Visualization for Memory Usage Analysis

As was illustrated in Chapter 2, the published works that propose software visualizations employ 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 [22, 27, 46, 54, 93, 156]. 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 [39, 40, 156]. Also, showing this information is helpful for data structure analysis by locating objects shared by data structures [9].

Although Vismep does not introduce a novel visualization technique, it adequately combines demonstrated techniques, such as the node-link diagram and the scatter plot. Also, it connects the source code with information from program execution, something that most visualizations dismiss [23]. Besides, Vismep does not explicitly show the allocations made during program execution; instead, it highlights the memory usage per function and line-by-line. Consequently, the level of detail of the information provided on memory consumption differs from that used by most visualizations. Furthermore, this study presents Vismep that supports visualizations of Python applications, which despite Python’s popularity, it is not often investigated in the published articles.

### 3.2.3 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 [27, 32, 46, 54, 123], showing the benefits of approaches. Nevertheless, little is known about how diverse programmers use and perceive visualizations to inspect applications’ memory usage. According to Chapter 2, 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 [22, 63, 156]. 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.

## 3.3 Vismep

In this section, we describe the design of Vismep, how it works, and illustrate the features with an example.

### 3.3.1 Overview

Vismep is an interactive visualization prototype designed to help programmers in analyzing the memory usage of Python applications. Vismep collects the memory traces during the execution of a Python program and displays the information through interactive views (see Figure 3.1). In addition, Vismep is equipped with an event-tracking system to facilitate the detection of events made by a user.

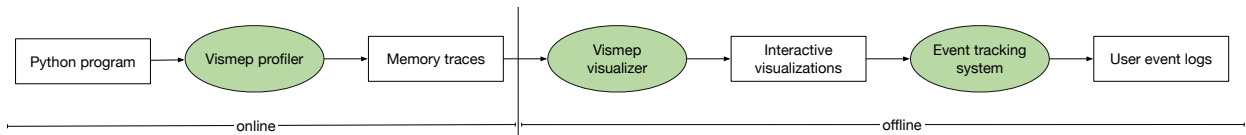


Figure 3.1: The Vismep overview. The Vismep profiler collects the memory traces from the execution of a Python program and generates CSV files with the information. Our visualizer then reads the files generated and displays the main interactive view (Call graph view) through which the user can navigate and access the other complementary views.

**Vismep profiler.** To extract run-time information of the program execution, we implemented a profiler for Vismep. This profiler is based on two popular python modules, *memory\_profiler*<sup>3</sup> and *trace*<sup>4</sup>. The Vismep profiler collects the following data:

- *Invoked functions/methods.* Vismep extracts a set of invoked functions/methods during program execution using the *sys* module<sup>5</sup>. For each function, it also collects: the function/method name, file, number of lines of code, and number of executions.
- *Memory usage per function and line-by-line.* Vismep gathers a memory footprint for each invoked function/method, such as the amount of memory allocated per function and line-by-line during program execution. The profiler collects this information based on the functionality of *memory\_profiler* module. In summary, we consider two aspects: (i) the memory usage of the Python interpreter after a line has been executed and

<sup>3</sup>[https://github.com/pythonprofilers/memory\\_profiler](https://github.com/pythonprofilers/memory_profiler)

<sup>4</sup><https://github.com/python/cpython/blob/3.10/Lib/trace.py>

<sup>5</sup><https://docs.python.org/es/3/library/sys.html>

(ii) the difference in memory of the current line with respect to the last one. We also consider this approach when extracting the memory usage per function.

- *Relationships between functions/methods.* The profiler collects the calling relationships between functions/methods during program execution based on the approach of *trace* module.

The profiler generates CSV files with the information previously mentioned. Note that Vismep extracts this information since our goal is to explore the usefulness of displaying aspects from the control flow and connecting the source code with dynamic information when programmers analyze memory usage. In Chapter 2, we reported that visualizations do not commonly display these aspects but some studies claim that it may be useful to comprehend memory management and detect the causes of memory anomalies. Unfortunately, these studies lack or present limited empirical evidence to show when or how these aspects are required. Furthermore, the profiler provides various helpful features of two popular modules previously mentioned. However, the profiler could also present some issues involved with these modules, such as the accuracy of memory usage, overhead, the performance execution, among others. In this chapter, we evaluate whether or not Vismep features are useful for programmers when analyzing memory usage. In addition, we also pinpoint aspects to improve the support.

***Vismep visualizer.*** To visualize the information extracted by the profiler, Vismep use polymetric views [74] to provide interactive visualizations that run on top of Pharo<sup>6</sup>, the live programming environment. We present four visualizations: (i) *Call Graph view*, (ii) *Source code view*, (iii) *Sub call Graph view*, and (iv) *Scatter plot view*. These visualizations and the interactions are detailed in the following subsections.

***Event tracking system.*** We equipped Vismep with an event-tracking system to facilitate the detection of events made by a user. This system automatically detects motions (*e.g.*, mouse hovering over a visual element, clicks) when the interactive visualizations are displayed. Vismep also presents a *End Session button* on the *Call graph view* to stop the tracking system and export the user actions to a CSV file. Vismep generates a new CSV file each time that the visualizer is opened.

The event tracking system generates a CSV file with the following information:

- *Time and position.* The current time and mouse cursor position.
- *View used.* The view in which the mouse cursor is located (if it is outside any view, it does not detect movements): Call graph, Scatter plot, Source code X, and Sub Call Graph X, where X is the selected function/method.
- *Event performed.* The events made by the user: over (by default, when the mouse cursor is above a visual element) or click (when the user selects a visual element).

---

<sup>6</sup><http://pharo.org>

- *Visual element.* The visual element involved with an event performed by the user: nothing (default value) or the name of the function/method that was focused (over) or selected (click).

During the data collection, we used this system to identify the actions made on a view, the popular views, and the flow followed to obtain the information that participants considered valuable.

The implementation of Vismep and the learning material are publicly available<sup>7</sup>. Additionally, we present an artifact that contains all the documentation necessary to install and execute Vismep in three different operative systems available on Zenodo and accessible through DOI [48].

### 3.3.2 Vismep In a Nutshell: Exploring a Pandas Issue

To illustrate Vismep, we analyzed a memory issue reported in the pandas package<sup>8</sup>. 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 listed in the righthand panel of Figure 3.2 (labeled as 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 3.2 gives an overview of Vismep. The left view 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* is displayed on the right-hand side to show the source code of the selected function (`export_dataframe`). Vismep also provides alternative views, such as *Scatter plot view* and *Sub call graph view*. Figure 3.5 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 3.4 displays the *Sub call graph view* that indicates the callers and callees of a particular function, in this case the `to_json` function.

### 3.3.3 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 [23]. It also displays the memory footprint and additional information (*e.g.*, name, number of executions, size) for each executed function/method. In Chapter 2, we discovered several alternatives to visualize information of calling relationships between functions. However, we noticed no empirical evidence of which visual representation is better for this purpose. We decided to use a node-link diagram because it is a visual representation that (i) is widely

<sup>7</sup><https://github.com/Balison/Vismep>

<sup>8</sup><https://github.com/pandas-dev/pandas/pull/45489>

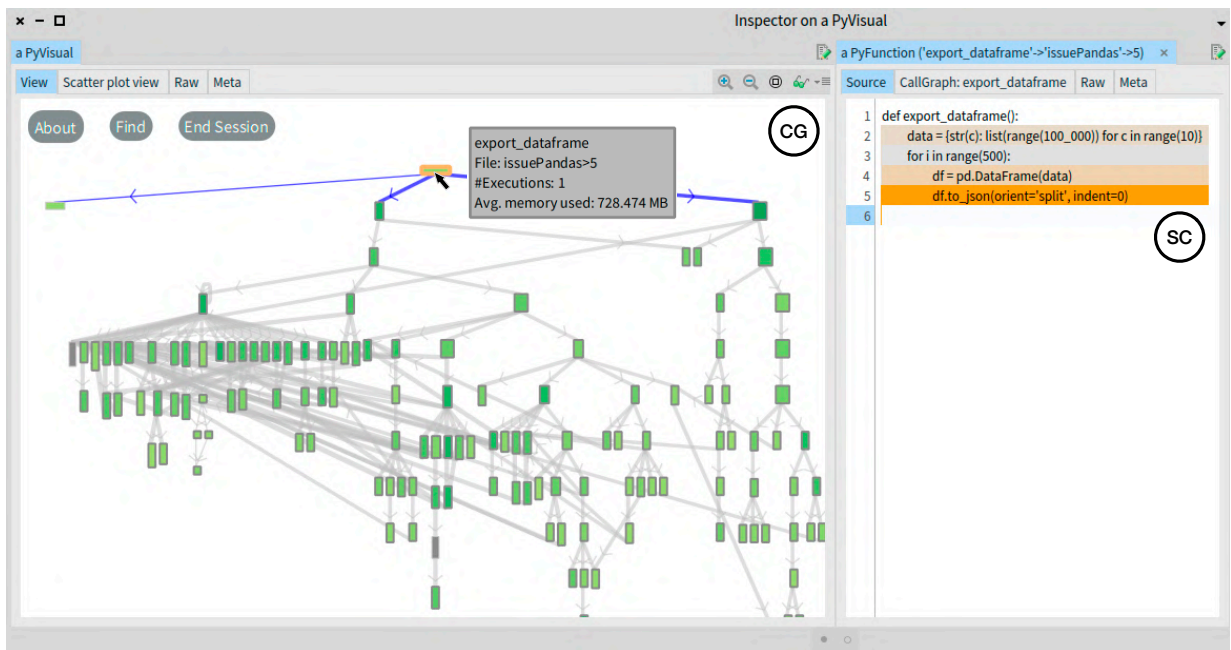


Figure 3.2: Illustrating a memory issue of pandas with Vismep. *Call Graph view* (CG) shows the memory usage along the execution path. Each node denotes an executed function, and the edges indicate the calling relationships. When a function is selected, its border turns orange, and its source code is 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.

utilized by current tools and (ii) is simple to understand and does not imply a very high learning curve [14].

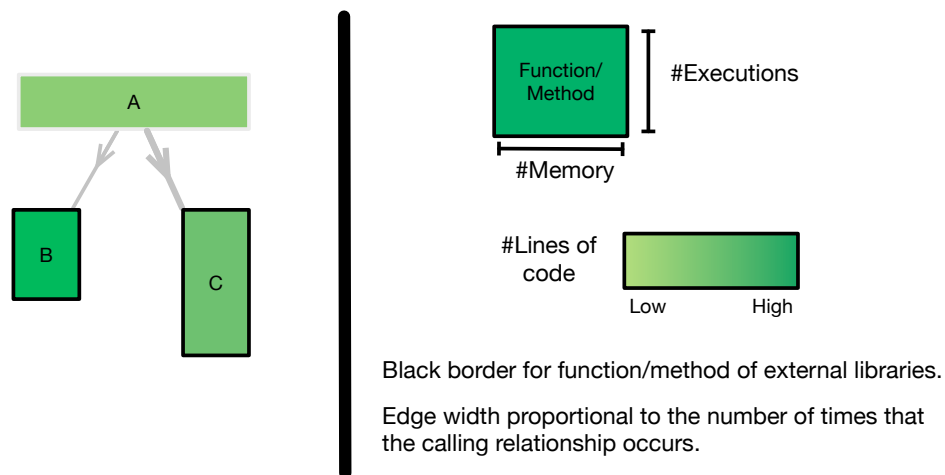


Figure 3.3: On the right, the legend for *Call graph view* and *Sub call graph view*, the width of a node corresponds to the function’s average memory, and the height denotes the times a function is executed. On the left, an example with *Call graph view*, where A function is executed few times and consumes a lot of memory. A calls first B (a few times) and then C (several times).



**Nodes.** As Figure 3.3 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.
- 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`<sup>9</sup>, the color is gray.
- The border shows if the function/method belongs to an external library (*e.g.*, pandas, random).

Note that the use of polymetric views [74] helps users identify exceptional entities (*e.g.*, hotspot allocations, unexpected memory usage) based on the visual attributes [42]. 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 3.2.

**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 3.3 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.

### 3.3.4 Source Code View

When a function/method is selected, a *Source code view* is built at the right, as shown in Figure 3.2 (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 [23]. To illustrate, in Figure 3.2, 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.

---

<sup>9</sup><https://docs.python.org/3/library/inspect.html>

### 3.3.5 Sub Call Graph View

Literature [14, 19] reported that the larger the node-link diagram, the more complex it is to understand the visualization. Consequently, we opted for creating the *Sub call graph view* to reduce complexity of exploring the *Call graph view*. Therefore, *Sub call graph view* 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.

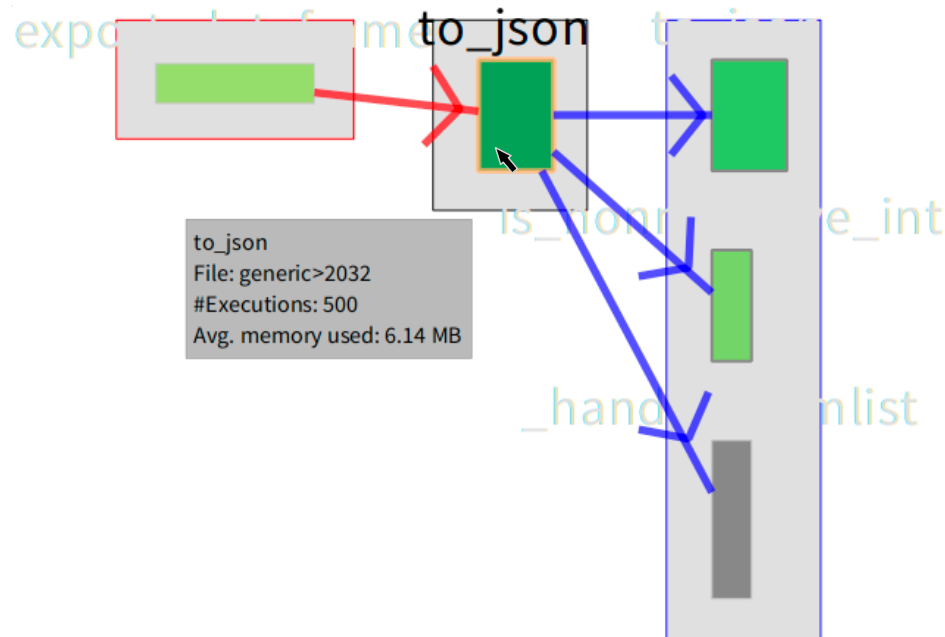


Figure 3.4: Overview of Sub call graph view.

Figure 3.4 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 3.4 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 3.4.

### 3.3.6 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.

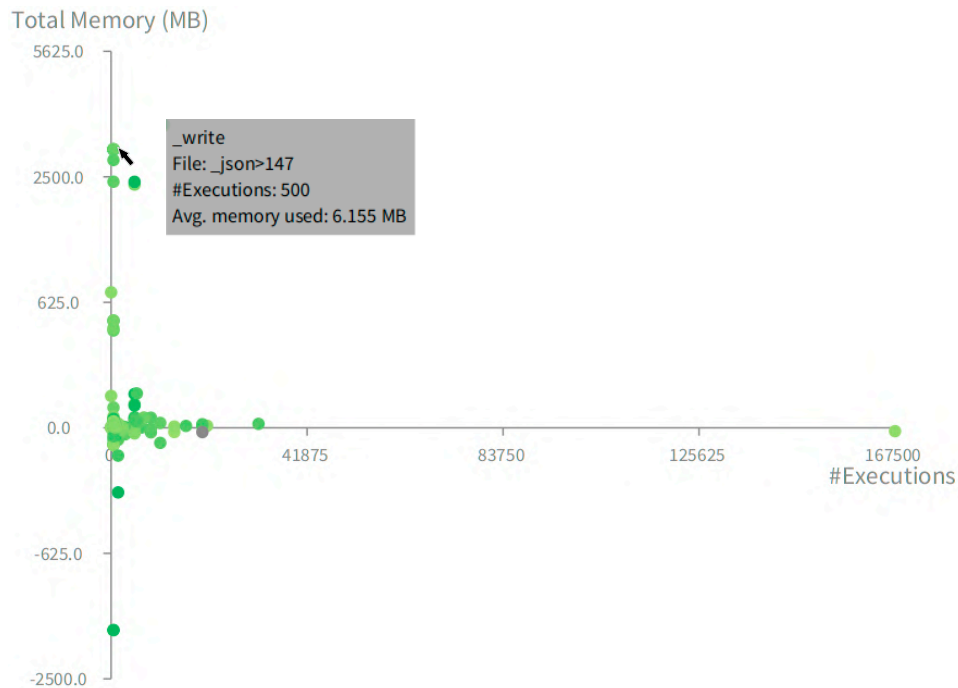


Figure 3.5: Overview of Scatter plot view.

Figure 3.5 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 3.2), and focuses on converting the `df` to a JSON string. More specifically, `_write` function allocates around 6 MB each time it is executed. Figure 3.5 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.

### 3.3.7 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 3.2 and Figure 3.4 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 3.2) and the *Sub call graph view* (Figure 3.4), 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 button named *Find* at the top left of the *Call graph view* (Figure 3.2) 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 3.2) and *Sub call graph view* (Figure 3.4). In addition, the user can navigate over these views by continually selecting nodes in each view, as illustrated in Figure 3.6. In this case, the user could observe the overview of the program with the *Call graph view* (left), inspect the source code of `export_dataframe` function (middle), and explore the calling relationships of `to_json` function (right) that is called by `export_dataframe` in line 5.

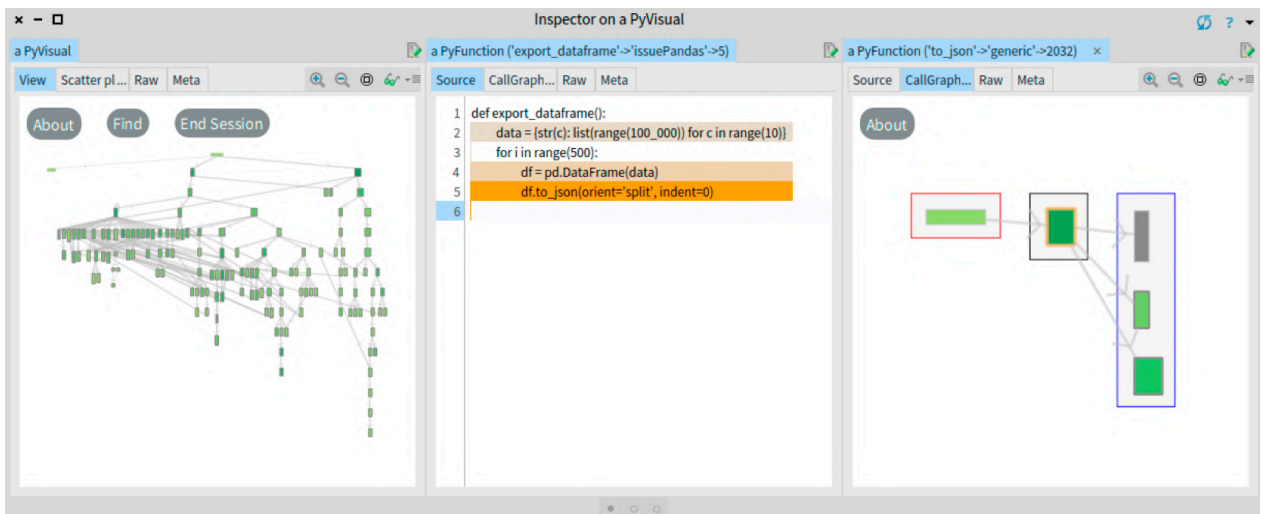


Figure 3.6: The user can navigate over different views of Vismep by continually interacting with each view. On the right, the user selects the `export_dataframe` function in *Call graph view*. Due to drill down, `export_dataframe` code is displayed in the middle view (*Source code view*). Then, the user selects the `to_json` called by `export_dataframe` to inspect the calling relationships of `to_json` function using *Sub call graph view*.

## 3.4 Methodology

We designed and conducted an exploratory study to understand how programmers employ and perceive Vismep when analyzing the memory usage of Python applications. Figure 3.7 illustrates the steps we used for our study.

The following subsections explain these steps.

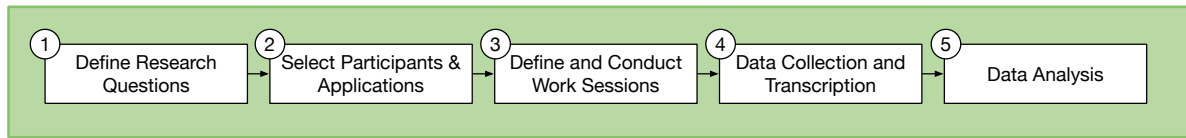


Figure 3.7: Overview of the workflow of the exploratory study.

### 3.4.1 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.

### 3.4.2 Participants & Applications

We invited students and bachelors from our university and members of Python communities to participate in our study. As a result, eleven programmers freely opted to participate (see Table 3.2), 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 were 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. We included Python programmers with different study fields since we consider that several Python users (*e.g.*, data analysts) do not necessarily have or are not pursuing a Computer Science degree.

Participants exhibited various levels of experience in Python programming. Their average experience in Python was 4.6 years (std. dev. 2.4). Participants also self-assessed their expertise using a Likert scale of five steps *i.e.*, 1 (novice) to 5 (expert). The average experience in Python programming was 3.4 (std. dev. 0.8).

Table 3.2: Information of participants (Python programming experience (years); Self-assessment expertise (Likert-scale: 1 (novice) to 5 (expert)); Experience in memory usage analysis; Experience in addressing memory issues; Strategies when analyzing memory usage).

ID	Study Field	Python Programming		Experience in Memory Usage Analysis and Issues		
		Experience (Years)	Self-assessment Expertise	Memory Usage Analysis	Addressing Memory Issues	Strategies Used
P1	Geology	9	2.5	✓	✓	Logs
P2	Computer Science	1.5	3	✓	✓	Logs
P3	Electrical Engineering	4	3.5	✓	✓	Manual
P4	Electrical Engineering	8	5	✓	✗	-
P5	Aerospace Engineering	2.5	3	✓	✗	-
P6	Computer Science	6	4	✓	✓	Manual
P7	Physical Engineering	5	3	✓	✓	Manual
P8	Mathematical Engineering	3	3.5	✓	✗	-
P9	Metallurgical Engineering	1	3	✓	✓	Manual
P10	Computer Science	4	2.5	✓	✓	Web search
P11	Computer Science	5	4	✓	✓	Manual

***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 participating in this study must choose a Python application to analyze during the study since monitoring memory usage is not a trivial activity, and the practitioner requires deep knowledge about the code under analysis. Consequently, participants selected different programs with which they were familiar. Most of the selected applications belong to the domain of data analysis, artificial intelligence, and machine learning. Additionally, they mentioned that their selection was based on either (i) they considered memory usage a potential threat to their application or (ii) they wanted to verify assumptions about memory usage and find ways to reduce memory usage.

### 3.4.3 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 study’s objective and characteristics described in the invitation to programmers who agreed to participate. The moderator also asked the participant to use the think-aloud technique [60] during the session.

Category	Question	Rationale
Characterizing memory usage	<b>Q1:</b> <i>Can you characterize the memory consumption of your application?</i>	The participant identifies and describes the information relevant to the memory usage analysis ( <i>e.g.</i> , allocation sites, allocations made).
Understanding memory usage	<b>Q2:</b> <i>What have you learned from your application? Do you find anything surprising (e.g., anomalies)?</i>	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	<b>Q3:</b> <i>Do you find an opportunity to decrease memory consumption?</i> <b>Q4:</b> <i>If you find an opportunity to decrease memory usage, can you improve it and run the profiler again?</i>	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 3.3: Questions answered by the participants.

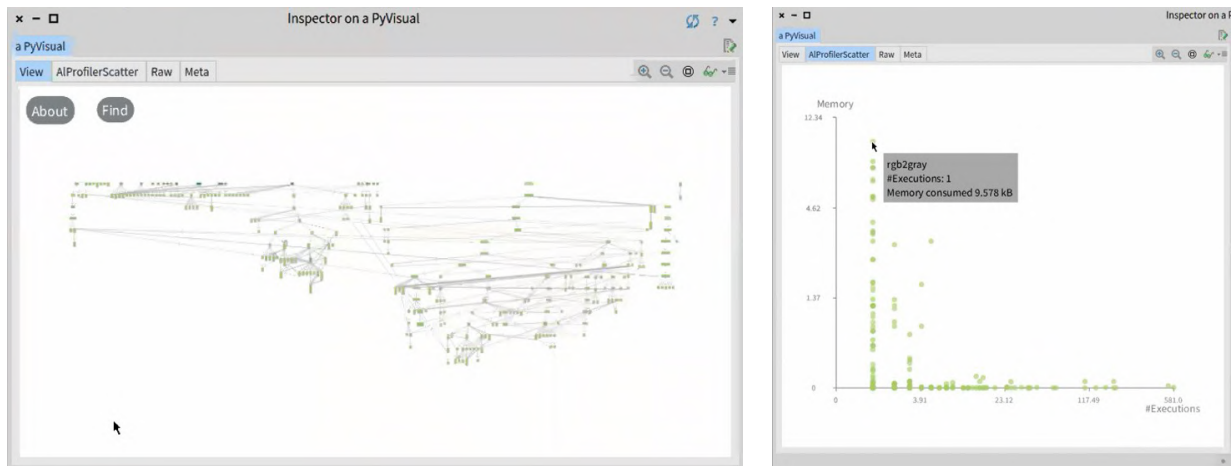
Additionally, each work session is structured as follows:

1. *Background and expectations.* The participant answered general questions to gather demographic data such as their age, gender, level of experience in Python programming, analyzing memory usage 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 read 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 3.3. We deliberately asked these questions to ensure that participants had a goal they cared about and looked for the information they considered valuable to understand memory usage and detect optimization opportunities. We consider that providing defined tasks (*e.g.*, select allocation hotspots) instead of these questions would prevent participants from naturally defining a goal they care about when analyzing memory usage.

4. *Online forms.* The participant filled out two online forms to measure the mental workload (NASA-TLX) [56] and the perceived usability of Vismep (SUS) [12]. These two self-assessment 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 the work sessions. We also recorded a video of the screen and the audio of the laptop used by the participants.

**Pilot study.** We conducted pilot studies with a set of other participants than those of the group that participated in the study. Our pilot studies involve three participants from computer science and three from mathematical and mechanical engineering. These participants have three to seven years of Python programming experience and a self-rated experience of between 3 and 4 using a five-step Likert scale, *i.e.*, 1 (beginner) to 5 (expert). During the pilot, each participant selected an application with familiar code (*e.g.*, own code, company project, personal project) and analyzed the memory usage of their selected application with Vismep. Figure 3.8 and Figure 3.9 illustrate screenshots of some pilot studies where participants explored the Vismep views to understand how the program manages the memory and runs.



(a) Call graph view

(b) Scatter plot view

Figure 3.8: During the pilot study, the first participant used the *Call graph view* (a) and the *Scatter plot view* (b) to identify the allocation hotspots.

Pilot studies helped us improve the tutorial and question descriptions in Table 3.3. The pilot also revealed some bugs or inconsistencies (problems with interaction mechanisms) in Vismep, which were resolved for the exploratory study.



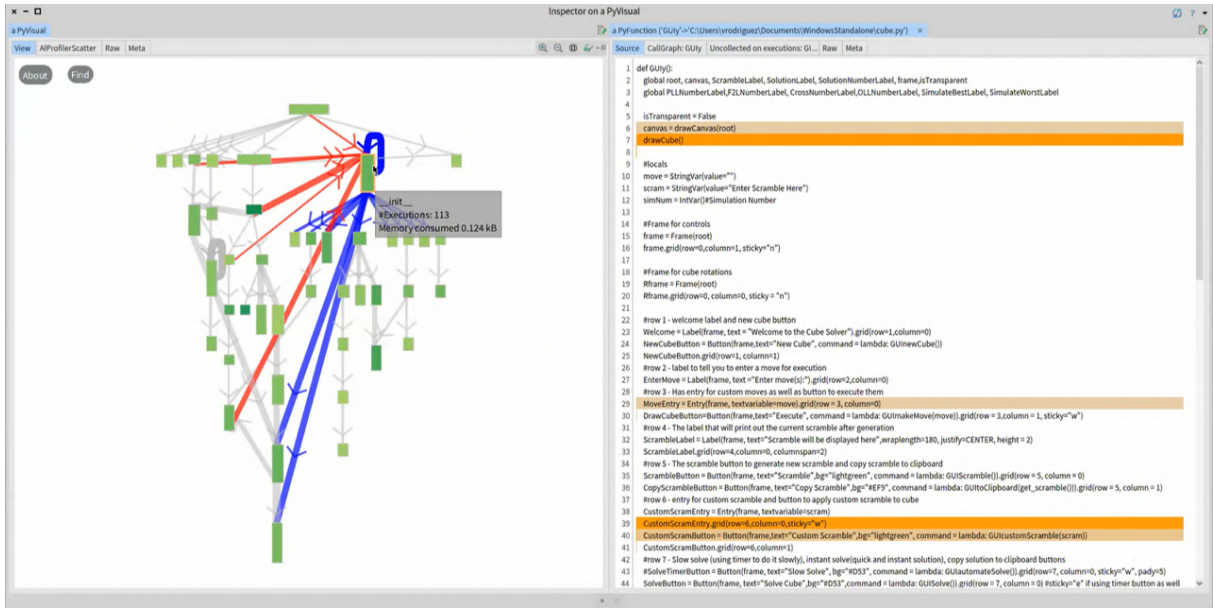


Figure 3.9: During the pilot study, the sixth participant explores the node’s source code in the *Call graph view*.

### 3.4.4 Data Collection and Transcription

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 3.3. We reviewed and checked the tracking logs and video recordings for each session to generate a spreadsheet that summarizes the session. 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 illustrate this information, consider the following example:

**Question asked:** Q1  
**Time:** 00:02:50 - 00:03:45  
**Verbalized thoughts:** The MAIN\_RUN function is the principal function of my program. What does the function do? This function initializes some variables and calls other functions to run the simulation and plot the data. Here, I also see that calling the CREATE\_REPORT function is the code line that consumes the most memory.  
**Participant actions:** The participant explores the source code of MAIN\_RUN function and describes what the code does by pointing out the lines responsible for calling other functions. The participant also locates the highlighted lines of the view and identifies the line that allocates the most memory.  
**Views used:** Source code view

Furthermore, to minimize biases during this process, the student supervised by the thesis

author generated the spreadsheets, and thesis author checked if the data was consistent with the audio, video records, and the logs from the event tracking system.

***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 [56]. 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 [12]. This questionnaire contains ten statements to measure the perceived usability of a system.

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

### 3.4.5 Data Analysis

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

***Interactions analysis.*** The thesis author analyzed the spreadsheets using open and descriptive coding [122] to identify themes (activities) related to the information required by participants (RQ1.1), similar to the study of Velez and colleagues [150]. One of the thesis supervisors checked the consistency of codes and observations to minimize biases during this process. Next, based on the generated codes, the thesis 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 [35] 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.

## 3.5 Results

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

### 3.5.1 RQ1.1: Information needs

We identified that participants looked for dynamic and static information for five themes to answer questions (Table 3.3). Table 3.4 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.

Table 3.4: Information needs, actions performed, and views (CGV = Call graph view; SCV = Source code view; SPV = Scatter plot view; SCGV = Sub call graph view) in Vismep that the eleven participants explored to analyze applications from Python.

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	Search functions/methods based on name.	CGV	11
			Search functions/methods based on module.		
			Inspect the functions/methods source code.	SCV	4
Allocation hotspots	Detect code that allocates most memory.	11	Discover and compare memory usage of functions/methods.	CGV	11
				SPV	9
				SCGV	2
			Discover and compare memory usage of code lines.	SCV	11
Circumstances, rationale and events	Understand under what circumstances functions/methods are executed, their rationale and the memory events related.	11	Inspect the functions/methods callers, callees and execution path.	CGV	11
				SCGV	7
			Inspect functions/methods rationale.	SCV	11
			Inspect functions/methods memory events (allocations, accesses, releases)		
Memory anomalies	Locate code involved with excessive or inefficient memory usage.	11	Analyze memory usage of allocation hotspots or relevant code.	CGV	11
				SPV	3
			Inspect the circumstances, rationale and events of allocation hotspots or relevant code.	CGV	7
				SCV	5
Anomalies cause	Locate the root cause of an anomaly.	7	Inspect the circumstances, rationale and events of memory anomalies.	CGV	7
			Analyze how memory anomalies affect the memory usage and functionality of relevant code	SCV	4

*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.

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.

### 3.5.2 RQ1.2: Use of Vismep

Table 3.4 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. Figure 3.10 illustrates a screenshot when the first participant detected allocation hotspots using *Call graph view*.

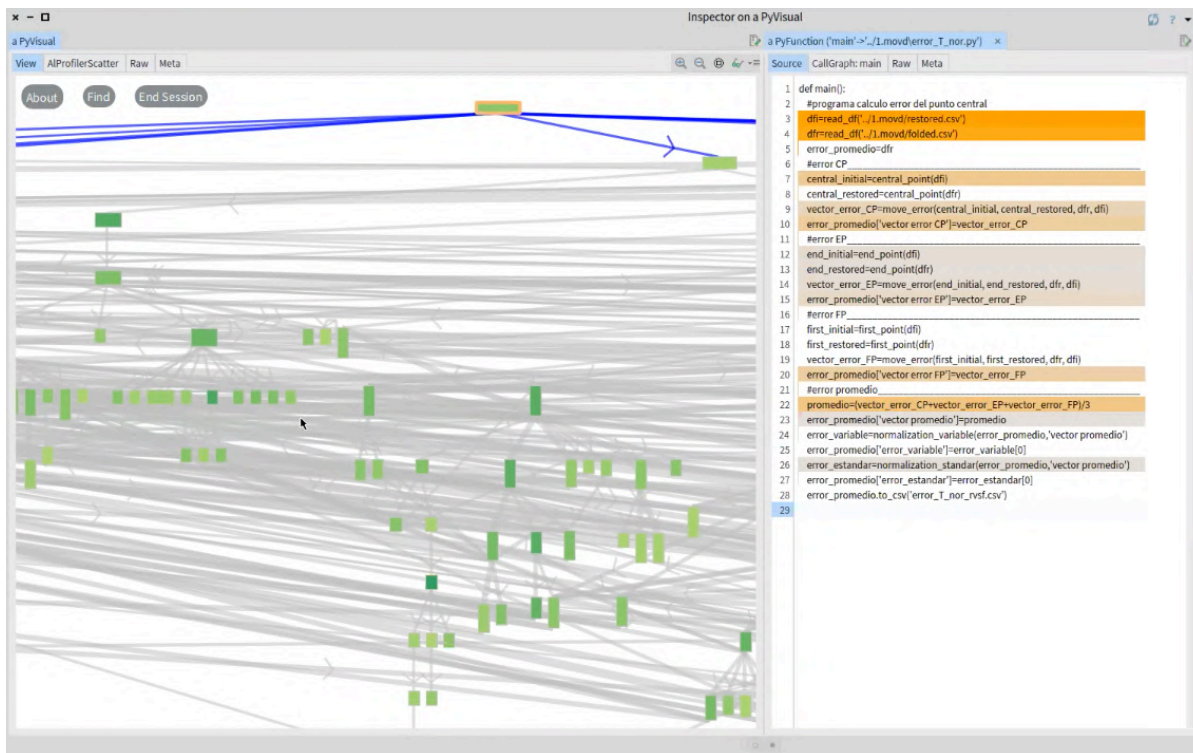


Figure 3.10: The first participant focused on `main` function in the *Call graph view*. Then, she selected the respective node to understand the code of `main` function and detect lines of code responsible for allocating the most memory.

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. To exemplify, Figure 3.10 displays a screenshot when the first participant selected a node corresponding to `main` in *Call graph view* and inspected the *Source code view* to understand the rationale of the code and memory allocations made. 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 by **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).

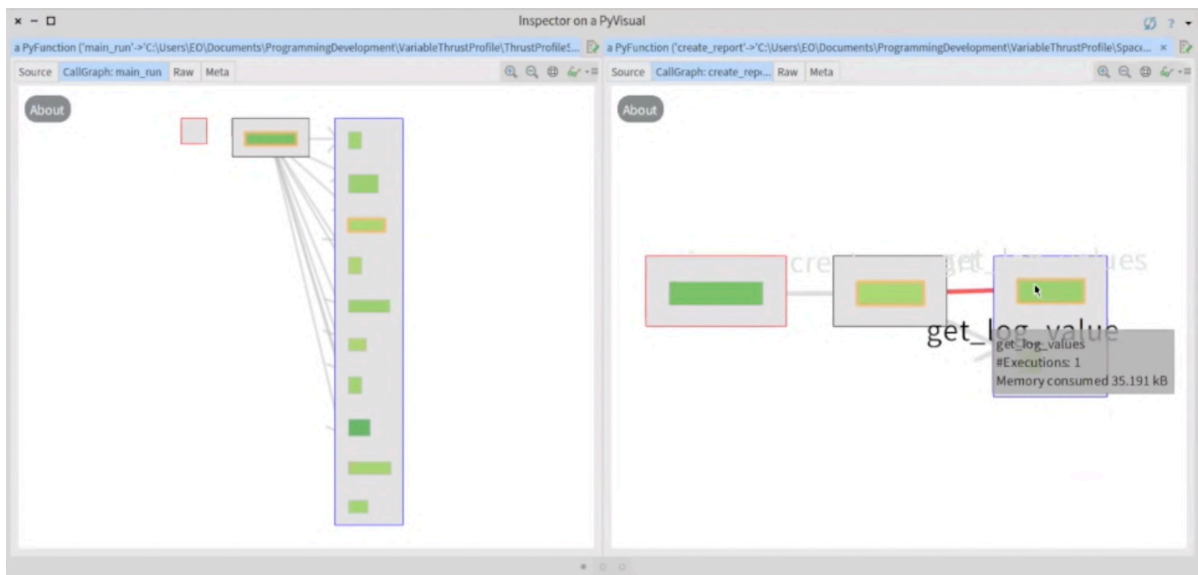


Figure 3.11: The fifth participant explored the calling relationships involved with `create_report` in *Sub call graph view* to determine under what circumstances this function is executed and if the memory allocated by this function is necessary.

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* or *Sub call graph view*. For example, Figure 3.11 illustrates when the fifth participant navigates through the calling relationships of `create_report` function to understand why this function is executed and if the memory used by this allocation hotspot is essential for the program. 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*.

### 3.5.3 RQ2.1: Cognitive Load

Table 3.5 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 [55] and Hertzum [57], this indicates a low to moderate effort.

Table 3.5: Ranges and means of overall workload and dimensions TLX scores.

	Min	Max	Mean	SD
<b>Overall TLX</b>	11.66	56.66	29.69	14.07
<b>Dimensions</b>				
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

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 3.5.5.

**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.

### 3.5.4 RQ2.2: Perception of usability

Table 3.6 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 [125] Vismep is graded “C+” which indicates a “good” usability score.

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



Table 3.6: Ranges and means of overall SUS and components of SUS scores.

	Min	Max	Mean	SD
<b>Overall SUS</b>	60	82.5	72.5	7.98
<b>Usability aspects</b>				
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

in positive aspects and lower scores for negative aspects (except for the need of support to use Vismep). Section 3.5.5 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.

### 3.5.5 RQ2.3: Perception of Vismep features

We identified nine general themes by using grounded theory [35] to process the feedback. Each theme is presented with its name, the number of times a theme occurs in the sessions and the number of participants who explicitly expressed it. First, we consider the following themes as positive:

- *Useful views and interactions.* [24 occurrences / 11 participants] 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 occurrences / 8 participants] 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 occurrences / 7 participants] 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 occurrences / 5 participants] 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 occurrences / 10 participants] 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 occurrences / 6 participants] 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 occurrences / 4 participants] 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 occurrences / 3 participants] 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 occurrences / 2 participants] 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.

## 3.6 Discussion

This study shows that our participants looked for 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.

It is expected that participants require dynamic information, especially about memory allocations, since it is commonly reported by tools/libraries in Python and software visualizations in general (see Chapter 2). However, the level of granularity in the information (*e.g.*, memory used, objects allocated, number of objects allocated) and the precision of this information varies between the diverse tools/libraries in Python and the software visualizations in general. During the work sessions, we observed that participants sometimes needed different levels of granularity in the information, including data more specific than the ones provided by Vismep or other profilers. For instance, Vismep only reports the memory usage per function and line-by-line, but it does not inform about the allocated data or the number of instances. When some participants tried to optimize the memory usage of their applications, they looked for information about specific allocations (*e.g.*, arrays, dictionaries). Although they could find the information through manual searches in Vismep, this process could be facilitated by showing different levels of granularity in the information. Therefore, we consider the granularity level in the information an interesting point to consider when supporting programmers.

Furthermore, some participants looked for information that Vismep does not provide, such as memory usage over time or explicit information about memory releases (information about garbage collection). For instance, a number of participants looked for information about the allocation and release of objects over time to detect the presence of temporary allocations (allocations created and released from memory several times) and reduce the workload in the garbage collector. Therefore, this study also exposes the relevance of other metrics related to memory usage. In these cases, other tools may support programmers with this information (see Section 3.2); however, it cannot be assured whether the programmers would be able to use them properly or how effective the tools are in these cases.

We also found that participants inspected the source code several times when identifying relevant code or comprehending the rationale behind the code. Additionally, Vismep supports participants in detecting code lines that allocate the most memory and understands why the memory is allocated by connecting the code with dynamic information (*Source code view*). As a result, we detect that mapping the source code with dynamic information helps programmers perform tasks related to memory usage analysis. Several software visualizations, tools, and libraries that support programmers when analyzing memory usage do not provide support for viewing the source code or connecting the code directly with dynamic aspects (see Chapter 2). Consequently, the programmer must use any of these options and a source code editor. We cannot be sure that connecting source code with dynamic information like we do with Vismep is much more comfortable and better for programmers than using a tool and a source code editor simultaneously. However, we believe that it could affect the user experience of programmers. For example, connecting the source code with dynamic aspects may reduce the mental demand perceived due to the operations of managing different environments simultaneously, as well as, increase integrity about the tool's functions and the perception

that it is easier to use. A follow-up study is necessary to confirm these assumptions.

We should also mention that our results are linked to Vismep. Consequently, if the participants had used another tool (*e.g.*, other data displayed and other visual techniques used), the needs and the flow that the programmers follow could have been different. However, we believe that our results give a good insight into the behavior and needs of programmers when analyzing memory consumption. Likewise, we believe that the design of our study represents a stepping stone in the area, since we present a method to provide empirical evidence about how a visualization performs in practice.

### 3.7 Threats to validity

We identified and organized some threats to our research’s validity based on the work of Wohlin *et al.* [163].

**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 3.6) indicates a need for help when using Vismep.

To minimize the inconsistency during the data collection and transcription process for RQ1, the student supervised by the thesis author generated spreadsheets that summarize the work sessions. To generate these spreadsheets, she used tools to generate subtitle files from the recordings, which subsequently go through a process of revising and improving the parts that have not been well written. The YouTube Marks<sup>10</sup> tool was also used to facilitate the analysis of videos by placing tags and adding comments to those tags in order to divide and identify the parts of the sessions more easily. Additionally, the thesis author checked if the spreadsheets generated were consistent with the audio, video records, and tracking logs to minimize biases during the process.

For the data analysis of this study, the thesis author identified the information needs. However, identifying information needs based on a participant’s behavior can be inaccurate since they do not always verbalize their thoughts explicitly. To minimize the inaccuracy in the process, a thesis supervisor contrasted different scenarios and the use of Vismep to satisfy the information needs. This supervisor also checked if the information needs identified were consistent with the information in the spreadsheets.

---

<sup>10</sup><https://github.com/tinchodias/youtube-marks>

**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.

**External Validity.** The difficulty of carrying out this type of large-scale study restricts the generalization of our results. For example, transcribing and obtaining the information needed for a session requires considerable time. This study presents a number of sessions similar to some related works (*e.g.*, Fernandez [22] with eight sessions), and the participants' variability may represent this tool's end users. The generalization of results is limited due to the number of participants, the lack of diversity in tasks, and the duration of the sessions. In addition, involving participants who voluntarily participate in the study and choose applications with specific memory issues according to the study goals takes time and effort.

This study focuses on analyzing the memory consumption of Python applications. During the study, we try to cover various applications (*e.g.*, artificial intelligence, machine learning, data science, simulators) according to the choices of each participant (own code, own project). Furthermore, it is necessary to mention that more complex projects could cause more visual clutter or more issues collecting data because Vismep's performance is one of its most significant weaknesses, as mentioned in the Section 3.5.5.

Although the external validity is limited, we believe this study provides relevant evidence on the feasibility of Vismep in supporting programmers when analyzing memory consumption.

## 3.8 Summary

This chapter 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. In the following paragraphs, we summarize our observations about the (i) information that participants looked for during the study for analyzing the memory usage of their programs, (ii) the usage of Vismep to obtain the information that they considered essential, and (iii) their perception of the tool in terms of mental workload and usability perception.

**Information needs.** 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**.

**Vismep usage.** Participants used different Vismep views or combined some of them to obtain the required information. 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 to use Vismep to perform the activities. We detected missing information that users required and possibilities to improve

the design of Vismep and other tools.

**Perceptions.** 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. Furthermore, we also detected positive and negative aspects that participants mentioned about Vismep. These aspects open the door to potential improvements and points to consider for supporting programmers, as well as, the factors (*e.g.*, easy to use, intuitive, useful) that participants desire from a tool.

In this chapter, we found that participants looked for dynamic and static information when analyzing memory usage. Consequently, displaying how the program runs (calling relationships between functions) and connecting the source code with dynamic aspects are considered beneficial. In the following chapter, we will look at the questions programmers ask while analyzing memory usage and how they answer those questions using Vismep and a popular Python package called Tracemalloc to analyze memory usage.

# Chapter 4

## Answering and Asking Questions During Memory Consumption Analysis

In Chapter 3, we reported the information that eleven programmers looked for when analyzing memory usage and how they used and perceived Vismep to obtain this information. In this chapter, we conducted a more exhaustive study to understand more precisely participants' questions when analyzing memory usage and how well Vismep and Tracemalloc address those needs. Therefore, we conducted an exploratory study with twenty-two programmers that assessed the memory consumption of their programs using Vismep and Tracemalloc. Note that in this study, we involved a greater sample of participants than in the study of Chapter 3, and we added Tracemalloc to provide more information about the needs of programmers when analyzing the memory consumption of programs and the use of tools. From our observations, we provide a catalog of 34 questions programmers ask themselves when analyzing memory consumption. We also present a detailed analysis of the use of Vismep and Tracemalloc to answer these questions and the difficulties that participants face during the process. Our results highlight the importance of (i) comparing and displaying information at different levels and (ii) connecting dynamic information with source code. As far as we are aware, our study is the first to highlight some challenges and express practicability concerns when analyzing memory consumption.

The content of this chapter is based on the publication "Asking and Answering Questions During Memory Consumption Analysis" (co-authored with Araceli Queirolo Cordova, Alexandre Bergel and Juan Pablo Sandoval) under review. This chapter is also involved with the bachelor's dissertation of Araceli Queirolo Cordova titled "Mejorar la usabilidad y efectividad de una herramienta de perfilamiento de memoria" [112]. As mentioned in Chapter 3, the bachelor's dissertation was supervised by the first supervisor and the author of this Ph.D. dissertation.

## 4.1 Introduction

Developers often spend a substantial amount of time manually monitoring memory consumption to localize memory anomalies (*e.g.*, memory leaks, memory bloats) that usually generate crashes on software applications [90, 139, 165]. For this reason, a number of memory profiling tools have been proposed providing a wide range of information displayed through full-text reports or visualizations [22, 27, 80, 154].

Although equipped with dedicated tools, previous work claimed that usually reported information may be insufficient or complex for programmers to locate memory anomalies and repair them [34, 78, 166]. Furthermore, other investigations also argue that how the information is displayed (full-text or visualizations) impacts the understanding of software analysis [43, 45, 103, 106].

There is still little understanding of the programmer’s needs when analyzing memory usage. For example, the questions *how does a programmer extract information to analyze the memory used by a program?* and *how well do current tools and approaches support this process?* have no precise and concrete answers, as mentioned in Chapter 2.

This chapter presents an exploratory study to provide a comprehensive and empirically-based set of questions that programmers ask during memory consumption analysis. We focused on understanding how programmers employ Python’s memory profiler tools because Python is considered one of the most popular programming languages<sup>1</sup>, and it is primarily applied to Data Science and Machine Learning applications in academia and several companies [88, 136]. The latter supports the intuition that Python programmers are more likely to face memory issues in their programming activities.

We selected two memory profilers, *Vismep* and *Tracemalloc*, for our study. These profilers provide diverse information through interactive visualizations and full-text reports, respectively. Memory profilers offer an extensive variety of features, and most profilers broadly differ on how information is provided and navigated. For example, some profilers [7, 8] may provide details about the garbage collector activity, while some others [5, 48] may focus on the context-call-tree or control flow. For this reason, selecting two different memory profilers instead of one hopefully enables us to cover different questions programmers ask themselves. In addition, the selected memory profilers together provide a variety of features typically proposed by current memory profilers.

We observed twenty-two programmers analyzing software applications with which they were familiar, using the two memory profilers, and responding to open questions. We deliberately asked open questions to ensure participants had a goal they cared about and looked for the information they considered valuable to understand memory usage and detect optimization opportunities. Then, we centered on collecting and analyzing data about the questions asked by participants and how they employed *Vismep* and *Tracemalloc* to answer those questions. For this, we followed a similar method of data extraction and analysis presented in other studies focused on identifying the information needs of developers [73, 133, 134].

---

<sup>1</sup><https://insights.stackoverflow.com/survey/2019>



This paper makes the following contributions:

**Catalog of questions.** We present an empirically-based catalog of 34 different questions asked by the participants. We have placed these questions into five categories based on the information needed and the behavior of the programmer to answer a question: understanding source code, understanding control flow, discovering the memory usage in a single point of time, comparing and contrasting memory consumption, and discovering memory events. To our knowledge, this is the most comprehensive list published to date that programmers may ask for when analyzing memory consumption.

**Tool usage analysis.** We provide an observational analysis about how programmers employ *Vismep* and *Tracemalloc* to answer the raised questions. We discovered that participants numerous times need to combine multiple views (*e.g.*, Call graph view and Source code view) from *Vismep* or use multiple API calls from *Tracemalloc* to obtain the required information. We also reported the questions that participants could not answer using these tools. Based on these results, we discuss the support missing from *Vismep* and *Tracemalloc*. Our analysis provides an opportunity to guide and improve the design of tools to answer particular questions and support programmers more effectively.

**Structure of the chapter.** Section 4.2 summarizes the prior empirical studies focused on gathering information needs in other fields. Section 4.3 details the exploratory study. Section 4.4 presents the 34 different questions and 775 question occurrences during the work sessions. Section 4.5 details how programmers employed *Vismep* and *Tracemalloc* to respond to the raised questions and which types of questions were not answered. In Section 4.6, we discuss our findings and the open challenges. Finally, we close the paper with threats to validity in Section 4.7 and conclusion in Section 4.8.

## 4.2 Related Work

Several studies centered on extracting information about developers' needs. As a result, these studies usually present emerging questions raised by developers in particular scenarios.

Sillito *et al.* detect 44 types of questions asked by programmers during software evolution tasks [133, 134]. These questions are categorized based on the kind and scope of the required information. Also, the study exposes that developers need better tool support to answer some specific questions. In a similar field, Kubelka *et al.* [72, 73] analyze the impact of live programming when developers perform software evolution tasks and identified eight additional questions compared to the set of questions provided by Sillito and colleagues [133, 134]. Additionally, Kubelka *et al.* notice that Live Programming impacts the questions asked by developers and the use of tools.

LaToza *et al.* survey professional developers and gather 94 questions grouped into 21 categories [76]. The study reports that the most frequent questions deal with the intent and rationale of code.

Fritz *et al.* identify 78 questions raised by developers with a lack of tool support during

software development [51]. The study also reports that answering these questions involves connecting information from different sources (*e.g.*, source code, change sets, teams).

De Alwis *et al.* collect 36 questions from literature, blogs, and their experience in software development [38]. They claim that developers present difficulties answering several questions because connecting multiple results from different tools is necessary to extract the required information.

Ko *et al.* detect 21 types of questions when analyzing software development teams [71]. The work highlights that the most frequent questions are related to mistakes in the code and co-workers' activities.

LaToza *et al.* focus on reachability questions and enlisted 12 questions with their difficulty and frequency [75]. The results show that reachability questions are challenging to answer and are associated with the most prolonged activities. Due to this, tools with support to answer these questions are relevant.

In contrast to the studies previously mentioned, to our knowledge, our work is the first observational study centered on developer information needs during memory consumption analysis.

**Impact.** We consider that documenting solid knowledge about programmers' needs while monitoring memory usage helps (i) improve the design and effectiveness of the current tools and new ones, (ii) recognize if a tool fits the programmers' needs and which needs may not currently be covered and (iii) facilitate the organization of current approaches to help practitioners find a suitable tool for their needs.

## 4.3 Methodology

Our study investigates the questions Python programmers raised during memory usage analysis and how programmers answer these questions using memory profiler tools. Consequently, we designed an exploratory study as Figure 4.1 illustrates. The following subsections describe the main steps of the study.

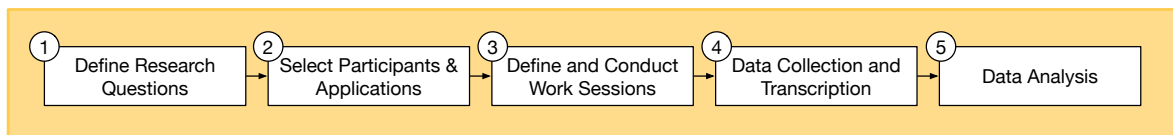


Figure 4.1: Overview of the workflow of the exploratory study.

### 4.3.1 Research Questions

We aim to answer the following research questions (RQ):

- **RQ1:** What questions do Python programmers ask when analyzing memory usage?
- **RQ2:** How do Python programmers answer these questions using Vismep and Tracemalloc?

To answer RQ1, we collected and classified questions asked by programmers during the work sessions. We inferred these questions based on the developer’s actions and events. Finally, to respond to RQ2, we analyzed the features of memory profiler tools that programmers employed to respond to questions identified in RQ1. We also detected the unanswered questions. In addition, we explored the efforts made by the participants to answer these questions and whether or not the tools used in this study were supportive in answering these questions.

### 4.3.2 Memory Profiler Tools

We selected Tracemalloc and Vismep to understand the impact of memory profiler tools on supporting programmers with memory usage analysis.

```

===== SNAPSHOT =====
Top 10 lines
#1: FindingEx.py:94: 140.1 KiB
    return tuple(map(lambda a, b: a + b, t1, t2))
#2: /Users/nosila/Documents/genetic_algorithm.py:115: 24.4 KiB
    return individual1[:_tmp] + individual2[_tmp:]
#3: FindingEx.py:119: 7.0 KiB
    return - math.sqrt(_dx * _dx + _dy * _dy) #- (_number_of_steps * 0.3)
#4: /Users/nosila/Documents/genetic_algorithm.py:89: 2.2 KiB
    return list(map(self.fitness_function, aPopulation))
#5: /Users/nosila/Documents/genetic_algorithm.py:61: 2.0 KiB
    new_population.append(self.create_new_individual(parent1, parent2))
#6: FindingEx.py:101: 0.9 KiB
    if(get_map(MAP, sum_tuple(_position, _d)) != WALL_CHARACTER):
#7: FindingEx.py:125: 0.9 KiB
    return [ gene_factory() for i in range(MAXIMUM_NUMBER_STEPS)]
#8: /Users/nosila/Documents/genetic_algorithm.py:30: 0.7 KiB
    self.breed_function = self.crossover
#9: FindingEx.py:143: 0.7 KiB
    ga = GA(pop_size=200, mutation_rate=0.1, fitness=fitness_distance_to_exit, individual_factor
y=path_factory, gene_factory=gene_factory, termination_condition=lambda f: f == 0, max_iter=5)
#10: FindingEx.py:144: 0.5 KiB
    best_fitness_list, avg_list, best_individual = ga.run()
29 other: 12.7 KiB
Total allocated size: 192.2 KiB

```

Figure 4.2: Enlisting the top ten memory allocation hotspots and total memory allocated using Tracemalloc.

**Tracemalloc.** As mentioned in Chapter 3, Tracemalloc is one of the most flexible libraries and provides multiple API calls to extract specific information related to memory usage through full-text reports. Programmers must modify the application’s code under study when using Tracemalloc [7]. Tracemalloc is part of the standard distribution of Python. It provides several features:

- *Display TOP.* This feature assists programmers in enlisting the memory allocation hotspots (code line, file), as shown in Figure 4.2.

- *Compute differences.* Tracemalloc supports programmers in exploring memory usage over time by indicating the differences (increase, decrease) in memory consumption before and after executing the potential leaking function.
- *Get traceback.* This feature helps programmers trace a particular memory allocation by identifying the execution path, specifically, in which case the memory allocation is made.
- *Get traced memory.* Tracemalloc presents functions that extract the total memory consumed and the memory used in maximum peeks.

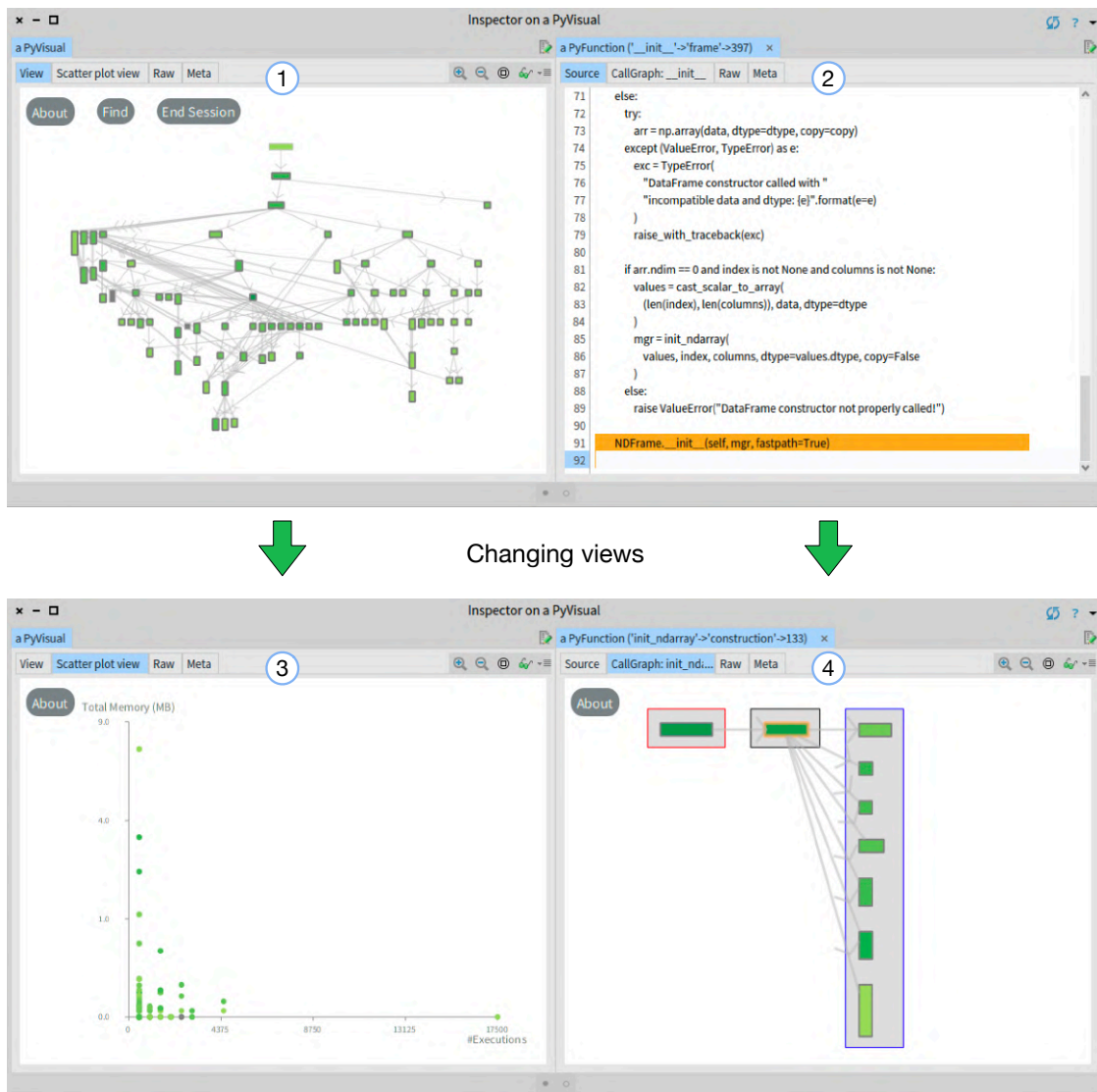


Figure 4.3: Visualizing an example with Vismep (1) Call graph view – the main view that summarizes the functions with the calling relationships and the memory footprint,(2) Source code view – a view that displays the source code and highlights allocation hotspots, (3) Scatter plot view – a secondary view that shows the relationships between memory used and the execution times of functions, (4) Sub call graph view – for navigating through the calling relationships of a function.

**Vismep.** As mentioned in Chapter 3, Vismep is an interactive visualization for supporting programmers in analyzing memory usage over Python applications. To employ Vismep, a developer must execute scripts with specific parameters. Vismep collects the invoked functions with their respective memory footprint during the program execution. Vismep also gathers the calling relationships between invoked functions. To display this information, Vismep provides multiple views as shown in Figure 4.3:

- *Call graph view.* The main view summarizes the functions with the calling relationships and the memory footprint using an interactive call graph. The call graph view assists users in locating relevant code, detecting allocation hotspots (functions), and identifying the circumstances in which memory is allocated.
- *Source code view.* This view displays the source code of a function and highlights the memory allocation hotspots (code lines). It supports practitioners in detecting allocation hotspots at a more fine-grained level (code line) and understanding the memory events involved with a particular code.
- *Scatter plot view.* Vismep provides a secondary view that shows the relationships between memory used and the execution times of functions. The scatter plot view assists programmers in exploring allocation hotspots (functions) and learning how the memory is used (allocated/released) during the program execution.
- *Sub call graph view.* Vismep facilitates users in navigating a selected function's direct calling relationships. As a result, the user can select a memory allocation site (function) and navigate through an execution path.

Vismep also provides several interaction mechanisms, such as canvas movement (*e.g.*, panning, zoom in and out), search option (find a function based on its name), and options to obtain detailed data about a particular function (*e.g.*, popup window with information-on-demand).

**Selecting memory profilers.** We selected Tracemalloc and Vismep for several reasons:

- *Reported information.* Tracemalloc and Vismep provide information to perform various tasks described in Chapter 3 (*e.g.*, analyzing allocation hotspots, analyzing the memory usage of entities). Furthermore, they connect dynamic information (*e.g.*, memory allocations) with the source code at different levels. Therefore, we consider that they help programmers trace memory events and better understand program behavior.
- *Report presentation.* Tracemalloc provides only full-text reports, which has become standard among other libraries and tools for profiling. On the other hand, Vismep reports the information only using interactive visualizations that could facilitate data comprehension [146, 159]. Consequently, we considered investigating how programmers employ both approaches.
- *Availability and maintenance.* Tracemalloc and Vismep are available, maintained, and provide material (structured documentation, examples) for practitioners to learn how to

use them. Tracemalloc is a native module<sup>2</sup> used internally in Python to improve other functionalities (*e.g.*, `ResourceWarning` reports). As a result, it works independently of the operative system and has no external dependencies. Furthermore, Chapter 3 introduced Vismep and presented how Vismep support programmers with memory usage assessment [48].

Therefore, we expect that selecting these two different memory profilers will increase the diversity and range of the questions asked by practitioners.

### 4.3.3 Participants & Applications

To recruit participants, we sent invitations to students and bachelors from our university and members of a number of Python communities. We make it clear in the invitation that the study investigates how programmers analyze the memory used by familiar code using memory profiler tools. We also explained that programmers who want to participate in the study must select a Python program. Since monitoring memory usage is not a trivial activity, participants must select an application they are familiar with (own code, project). Besides, they should be interested in the activity because (i) they considered memory usage a potential threat to their application or (ii) they wanted to verify assumptions about memory usage and find ways to reduce memory usage.

We selected twenty-two programmers who agreed to participate voluntarily and chose a Python application for the study. These participants belong to diverse study fields since we consider that several Python users (*e.g.*, data scientists, journalists) do not necessarily have or are not pursuing a Computer Science degree. We also included participants who were not necessarily experienced in dealing with memory issues since our goal is not to observe how participants specifically address particular memory issues.

***Demographic data.*** Twenty-two programmers participated in our study, of which six were females. Five participants were from the industry, four were in research centers, four pursued a master’s degree, and the rest were in their bachelor’s studies. In addition, a total of eleven participants have or are pursuing a degree in Computer Science, and the remaining in other fields (*e.g.*, Geology, Math, Electrical).

***Experience in Python programming.*** Participants manifested various experience levels in software development, but all were familiar with Python programming. Their average experience in programming with Python was 4.70 years (std. dev. 2.24). Also, participants self-assessed their experience using a Likert scale of five steps, *i.e.*, 1 (novice) to 5 (expert). As a result, the average experience in Python programming was 3.29 (std. dev. 0.73).

***Experience in memory usage analysis.*** All the participants have experience in monitoring memory usage and fourteen participants have experience addressing memory issues. We surveyed the participants about their common practices when analyzing memory consumption. Then we applied a coding technique [126] to study the answers in order to recognize the

---

<sup>2</sup><https://python.readthedocs.io/en/stable/whatsnew/3.6.html>

Table 4.1: Information of participants (Python programming experience (years); Self-assessment expertise (Likert-scale: 1 (novice) to 5 (expert)); Experience in memory usage analysis; Experience in addressing memory issues; Activities when analyzing memory usage). Participants from groups G1 and G2 present gray and white backgrounds, respectively.

ID	Study Field	Gender	Python Programming		Experience in Memory Usage Analysis and Issues		
			Experience (Years)	Self-assessment Expertise	Memory Usage Analysis	Addressing Memory Issues	Activities Performed
P1	Geology	Female	9	2.5	✓	✓	A1, A2
P2	Electrical Engineering	Male	4	3.5	✓	✓	A1, A2
P3	Electrical Engineering	Male	6	4	✓	✓	A1, A2, A3
P4	Physical Engineering	Female	5	3	✓	✓	A1, A2
P5	Electrical Engineering	Male	8	5	✓	✗	A1, A2
P6	Aerospace Engineering	Male	2.5	3	✓	✗	A1, A2
P7	Computer Science	Female	1.5	3	✓	✓	A1, A2
P8	Computer Science	Male	6	4	✓	✓	A1, A2
P9	Computer Science	Male	4	3	✓	✓	A1, A2
P10	Computer Science	Male	5	4	✓	✗	A1, A2
P11	Computer Science	Female	3	3	✓	✗	A1, A2
P12	Metallurgical Engineering	Male	1	3	✓	✓	A1, A2
P13	Electrical Engineering	Male	8	3.5	✓	✓	A1, A2
P14	Pedagogy in Math and Computing	Female	3	2	✓	✓	A1, A2
P15	Mathematical Engineering	Male	3	3.5	✓	✗	A1, A2
P16	Pedagogy in Math and Computing	Male	1	2	✓	✗	A1, A2
P17	Computer Science	Male	7	4	✓	✓	A1, A2
P18	Computer Science	Male	5.5	4	✓	✓	A1, A2
P19	Computer Science	Male	5	2.5	✓	✓	A1, A2, A3
P20	Computer Science	Male	6	4	✓	✓	A1, A2
P21	Computer Science	Female	4	3	✓	✗	A1, A2
P22	Computer Science	Male	6	3	✓	✗	A1, A2

practices adopted. We identified that participants often perform two or more activities:

- *A1: Analyze memory usage of entities.* All participants often examine the memory used by particular entities (variables, data structures, functions). For instance, P5 said “*I frequently explore how much memory allocates certain functions or data structures (arrays, dictionary) which I suspect could cause excessive memory usage*”.
- *A2: Analyze allocation hotspots.* All participants usually investigate the hotspots (code that consumes most memory) and which allocations are made. For example, several participants mentioned “*I usually focus on finding code that consumes most memory, and I try to understand why it allocates this amount of memory*”.
- *A3: Analyze memory usage over time.* Two participants usually explore how the memory is managed (used, released) over time. They mentioned that this action helps detect memory leaks and excessive memory usage.

We also observed that most participants usually modify their code to analyze allocation hotspots or the memory usage of specific entities (*e.g.*, variable, function) by using functions from dedicated libraries. Only one participant usually employs a memory profiler tool.

For our study, we randomly divided the participants into two groups, G1 and G2, each containing eleven participants. These groups are balanced with participants from the computer science field and other fields (*e.g.*, electrical engineering, mathematical engineering). Table 4.1

summarizes the profile of the participants and presents the participants from G1 and G2 with gray and white backgrounds, respectively.

**Applications.** As mentioned before, we explicitly asked participants to choose a Python application to analyze during the study. Since monitoring memory usage is not a trivial activity, we suggest that participants select an application they are familiar with (own code, project) and find it interesting to analyze in terms of memory consumption. Consequently, participants selected different programs (*e.g.*, data analysis, artificial intelligence, machine learning) with which they were familiar. Additionally, they mentioned that their selection was based on either (i) they considered memory usage a potential threat to their application or (ii) they wanted to verify assumptions about memory usage and find ways to reduce memory consumption.

### 4.3.4 Procedure

We conducted a work session for each participant with her/his respective application. Each work session started with a moderator explaining the study’s goals described in the invitation to programmers who agreed to participate in the study. The moderator also explained how the think-aloud protocol [60] works and asked the participant to use it during the session. Then, general questions are asked to the participant to collect demographic data such as age, gender, experience in Python programming, memory usage analysis, and addressing memory issues. After these questions, the participant describes her/his application and gives an opinion about the application’s memory usage. The participant also explains the expectations or assumptions about elements (*e.g.*, functions, instances) that may produce a memory anomaly (*e.g.*, memory bloat, leak) during the program execution.

Furthermore, the session proceeded with two phases, each with a different memory profiler tool. Both phases are structured as follows:

Table 4.2: Questions made to the participant.

Open Question	Rationale
<b>Q1:</b> <i>Can you characterize the memory consumption of your application?</i>	The participant identifies the information relevant for memory usage analysis, such as the allocation hotspots or the allocations made during program execution.
<b>Q2:</b> <i>What have you learned from your application? Anything surprising?</i>	The participant contrasts the profiler’s report with her/his assumptions. Also, she/he describes if the profiler’s report provides additional or unknown valuable information and the potential issues that the application has.
<b>Q3:</b> <i>Do you find an opportunity to decrease memory consumption? If yes, can you improve it and run the profiler again?</i>	The participant describes and discusses the opportunities to reduce the application’s memory usage, if any. The participant also modifies the code that could cause a memory anomaly. Then, the participant verifies the impact of the changes in the application’s memory usage using the memory profiler tool.

1. *Exploration.* The participant read the documentation about the memory profiler tool and had an exploration phase to familiarize herself/himself with the tool. The participant



also had the opportunity of asking the moderator questions about the tool or its documentation.

2. *Open-questions.* The participant employed the respective memory profiler tool for analyzing the memory usage of her/his application and answered the open questions in Table 4.2. Since our purpose is to explore how participants behave when analyzing memory usage, we deliberately asked open questions to ensure that participants had a goal they cared about and looked for the information they considered valuable to understand memory usage and detect optimization opportunities.
3. *Online forms.* The participant filled out two online forms to obtain information about the mental workload (NASA-TLX) [56] and the perceived usability (SUS) [12] for both tools. These two self-assessment techniques are hugely popular in empirical studies and are applicable to provide information about perceptions.
4. *Post-study questionnaire.* The participant answered verbally and informally open questions regarding their observations, recommendations, and perceptions of the memory profiler tool.

In the case of the participants in G1, they first worked with Vismep and then with Tracemalloc. For participants in G2, they first employed Tracemalloc and then Vismep. In both phases, participants analyzed the same application they selected. Note that our goal has not been to compare Vismep and Tracemalloc but to explore how programmers use these tools when analyzing memory usage.

We recorded a video of the screen and the laptop's audio used during the work sessions. These recordings were used later to collect data and analyze participants' reasoning process.

***Study Decisions.*** We asked the same open questions proposed in Chapter 3 since we aimed that participants had a goal they cared about and looked for the information they considered valuable to understand memory usage and detect optimization opportunities. Therefore, we considered that providing defined tasks (*e.g.*, select allocation hotspots) instead of these questions would prevent participants from naturally defining a goal they care about when analyzing memory usage.

Furthermore, we expected that dividing participants into two groups (G1 and G2) and analyzing the same application with Vismep and Tracemalloc would increase the diversity and range of the questions asked by practitioners. We also contemplated that programmers might analyze memory usage and address memory issues using several tools since they require various features not necessarily supported by a single tool, as shown in Chapter 3. Note that our goal has not been to compare Vismep and Tracemalloc but to explore how programmers use these tools when analyzing memory usage.

Finally, note that participants filled out two online forms to measure the mental workload (NASA-TLX) and the perceived usability (SUS) for both tools. We gathered this information to obtain feedback about participants' perceptions of the tools (see annex B) and consider it for future research.

### 4.3.5 Data Collection and Transcription

This section describes the variety of data gathered to answer our research questions and the data collection process.

**Events extraction.** To answer RQ1 and RQ2, we aim to identify questions asked by the practitioners and how they employ memory profilers during memory consumption analysis. To achieve our goal, we use a method similar to the Chapter 3 study to extract information from the work sessions. Therefore, we collected data when participants employed Vismep and Tracemalloc to answer the open questions in Table 4.2.

Firstly, we reviewed and checked session video and audio recordings to generate spreadsheets that summarize each work session. Each spreadsheet contains: (i) the memory profiler (Vismep or Tracemalloc) used, (ii) the open question that the participant responded, (iii) the respective period of time in the video record, (iv) the verbalized thoughts of the participant, (v) the actions made by the participant, and (vi) the memory profiler tool features used.

To minimize biases in the data collection process, the student supervised by the thesis author generated the spreadsheets, and the thesis author checked if the data from the spreadsheets was consistent with the audio and video records.

### 4.3.6 Data Analysis

This section describes the methods employed to analyze the collected data.

**Questions inference.** To identify the questions asked by the participants during our study, we performed an analysis method similar to the one proposed by Kubelka and colleagues [73]. This method consists of two steps: (i) identifying the concrete questions by analyzing the video and (ii) generalizing, encoding, and unifying the concrete questions.

Firstly, we detected concrete questions using the spreadsheets generated for each work session. Some questions were inferred based on the actions and verbalized thoughts of the participants. For example, we inferred “*What part (function, line of code, instance) of main function consumes the most memory?*” as the concrete question for the actions mentioned in the following example:

**Tool:** Vismep

**Open-question:** Q1

**Time:** 00:01:40 - 00:01:53

**Verbalized thoughts:** For the `main` function, the lines that consume much memory are reading the file, detecting the fire, and showing the points where the fire is detected.

**Participant actions:** The participant observes the source code of `main` function and jumps to the highlighted lines of the view while pointing out and describing the code lines.

**Features used:** Source code view

**Inferred question:** What part (function, line of code, instance) of `main` consumes the most memory?

We also gathered concrete questions that the participants directly mentioned. To illustrate, P6 passed the cursor over the `CALC_PARAMETERS` function and examined the connected nodes with blue edges (outgoing functions) while he asked, “*Which functions do `CALC_PARAMETERS` call?*”.

Then, we defined generic questions based on concrete questions to abstract the details of a given task. For this, we generalized the questions by identifying similar concrete questions. For instance, we inferred the generic question “*Which functions does this function call?*” from the concrete question “*Which functions does `CALC_PARAMETERS` call?*” to reference any function on the execution of a program. We also mapped some questions with the questions mentioned by Sillito *et al.* [133, 134] and Kubelka *et al.* [73] due to the presence of questions related to understanding the control flow and the rationale behind the source code. For example, we transcribed “*Where is this method called or type referenced?*” proposed by Sillito *et al.*, instead of our generic question “*Which functions call this function?*”. We noticed that the participants asked these type of questions to enrich their comprehension of the software application and make decisions about memory anomalies or opportunities for improvement in memory usage.

To minimize biases in the data analysis, the student supervised by the thesis author inferred the concrete and generic questions, and the thesis author checked if the inferred questions (concrete and generic) were consistent with the information in spreadsheets. In addition, the thesis author and the two thesis supervisors held two meetings to discuss any discrepancy in the inferred questions’ consistency.

***Classification scheme creation.*** We opted to organize the discussion of the inferred questions around a classification scheme. To create this classification scheme, the thesis author conducted a thematic analysis [141] by following these steps:

- *Familiarization.* The inferred questions were read and reread to obtain an overview of the data.
- *Generating codes.* To reflect relevant features of each question, the author in charge assigned codes. For instance, the author assigned the code “*Understanding intent and implementation*” for the question “*Which entities (functions, lines of code, instances) are*

*involved in the implementation of this behavior?*”. Furthermore, the author conducted continuous reviews to refine and check the consistency of the assigned codes.

- *Constructing initial themes.* The author created coherent groups to identify broader patterns based on the assigned codes with their associated inferred questions. If a code does not belong to a specific theme, it is assigned to a miscellaneous group and analyzed later.
- *Reviewing themes.* The author checked the initial themes against the inferred questions to refine and create the final themes.
- *Defining and naming themes.* Final themes were defined with a descriptive name and a detailed description.

Additionally, the two thesis supervisors reviewed the consistency of codes and themes against the associated data. We held two meetings to discuss the disagreements or potential issues of the generated codes and themes. Consequently, we minimized potential inconsistencies in the coding process.

Furthermore, a thesis supervisor and a professional software engineer independently categorized the inferred questions using the generated classification scheme to validate the reliability. For this, each one filled out a spreadsheet to categorize the inferred questions based on the detailed description of the created categories. Subsequently, we calculated the Cohen kappa [152] as a reliability metric to examine the agreement between reviewers. As a result, we detected that reviewers present an “almost perfect agreement” ( $kappa > 0.81$ ). The latter suggests that the classification scheme presents accurate representations (themes) for the inferred questions.

***Events and questions analysis.*** To answer RQ2, we first extracted the actions made by the participants and the features used to answer the inferred questions. Then we gathered patterns by concentrating on frequent actions performed to answer specific questions using certain features of Vismep and Tracemalloc. Furthermore, we also located questions that participants declared they could not answer using the memory profiler tools.

We considered whether or not the question asked was answered based on the verbalized thoughts of the participants. For instance, participants suggest that questions were not answered when they mentioned phrases like “*I can’t find out with the information I have*” or “*I think this change would reduce memory but I can’t do it right now because it will take a lot of time and it’s complex*”.

## 4.4 Results: What Questions do Python Programmers Ask During Memory Usage Analysis?

We identified 34 different questions and 775 question occurrences from 22 work sessions, with a total duration of 24 hours (exploration, open-questions, and post-study questionnaire). Also, we generated a classification scheme with five categories to organize these inferred questions.

Table 4.3 illustrates the distribution of questions per category and the occurrences raised by the participants during the work sessions. The first column provides the questions categorized according to our classification scheme. The following twenty-two columns summarize the questions occurrences during each work session with the format: #Total questions occurrences (#Occurrences using Vismep / #Occurrences using Tracemalloc) if any occurrence, otherwise the cell is empty. We also use the cell background to show the relative frequency of the questions, in which green darker backgrounds indicate the most frequent questions in the sessions. Additionally, the last column follows the format previously described and represents the total number of question occurrences per question or category.

The following sections describe each category and the questions related to them.

#### 4.4.1 Understanding Static Structure, Intent and Implementation

This category involves questions centered on understanding aspects of the source code, such as its static structure, rationale, and implementation. We noticed that in this category, six questions and 158 question occurrences (20%) were raised during the work sessions. The following questions are most frequent in this category: “4. *What does the declaration or definition of this look like?*”, “1. *Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?*”, and “2. *Which entities (functions, lines of code, class) belong to this file or module?*”

We observed that these questions usually were asked when participants: (i) searched for a particular piece of code to analyze its memory usage and (ii) located a memory anomaly or an opportunity to reduce memory usage.

***Finding a particular code.*** When participants characterize the memory consumed by their applications, they commonly focus on finding entities (functions, instances) relevant to the program functionality based on their knowledge to analyze its memory usage. For example, P3 wanted to analyze the memory usage of some auxiliary functions that he considered relevant: “*I want to find my auxiliary functions that perform multiple calculations and I want to know how much memory they consume*”. Participants usually found relevant entities based on the provided behavior, the module they belong to, and the entity’s name. As a result, the next questions were asked: “1. *Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?*”, “2. *Which entities (functions, lines of code, class) belong to this file or module?*”, and “3. *Is there an entity named something like this in that unit (project, package, or class)?*”

***Locating anomalies and improvements.*** When participants tried to determine if an entity (function, line of code) was involved with an anomaly (*e.g.*, memory bloat, leak), they often explored the entity’s source code to decide if the memory consumed was necessary or not for the correct functionality of the program. Furthermore, when participants locate an anomaly or an opportunity for reducing memory usage, they center on the parts of code responsible for the anomaly to investigate how it is defined or structured and what the code is supposed to do. Consequently, participants asked the following questions: “4. *What does the declaration or definition of this look like?*”, “5. *What are the parts of this entity (function,*

Table 4.3: Questions per category and the occurrences raised during the work sessions. Each column corresponding to a participant presents the format: T (A/B), where A denotes the number of occurrences using Vismep, B indicates the number of occurrences using Tracemalloc, and T denotes the total number of occurrences. Note that if T is zero, the cell is empty.

Questions Type per Category	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	P15	P16	P17	P18	P19	P20	P21	P22	Total
<b>Understanding Static Structure, Intent and Implementation</b>																							
1. Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?	2 (2/0)	1 (1/0)	2 (2/0)	2 (2/0)	1 (1/0)	1 (1/0)	2 (1/1)	1 (1/0)	1 (1/0)	2 (2/0)	1 (1/0)	2 (2/0)	2 (2/0)	1 (1/0)	2 (2/0)	2 (2/0)	1 (1/0)	2 (2/0)			3 (3/0)	2 (2/0)	33 (32/1)
2. Which entities (functions, lines of code, class) belong to this file or module?	4 (1/3)	1 (0/1)		3 (2/1)	2 (0/2)	2 (1/1)	1 (1/0)	2 (1/1)	3 (2/1)			3 (2/1)	2 (2/0)	2 (2/0)	2 (2/0)		1 (1/0)	2 (1/1)	1 (0/1)		4 (3/1)		33 (19/14)
3. Is there an entity named something like this in that unit (project, package, or class)?			1 (1/0)	2 (2/0)	1 (1/0)	1 (1/0)							6 (6/0)	2 (2/0)			2 (2/0)	1 (1/0)				1 (1/0)	17 (17/0)
4. What does the declaration or definition of this look like?	4 (0/4)	7 (1/6)	3 (2/1)	2 (2/0)	2 (0/2)	4 (2/2)	2 (2/0)	1 (1/0)	4 (3/1)			5 (5/0)	3 (3/0)		1 (1/0)				5 (1/0)	1 (1/4)	1 (1/0)	1 (1/0)	46 (26/20)
5. What are the parts of this entity (function, instance, type)?									2 (2/0)														2 (2/0)
6. What is the behavior that these entities (functions, lines of code, instances) provide together?	2 (1/1)				2 (1/1)	1 (1/0)	2 (2/0)	2 (2/0)	3 (1/2)	1 (0/1)		1 (1/0)	1 (0/1)		3 (2/1)		1 (0/1)		7 (2/5)		1 (1/0)		27 (14/13)
<i>Total in the category</i>	12 (4/8)	9 (2/7)	6 (5/1)	9 (8/1)	8 (3/5)	9 (6/3)	7 (6/1)	8 (7/1)	11 (7/4)	3 (2/1)	1 (1/0)	11 (10/1)	13 (13/1)	3 (3/0)	7 (7/1)	2 (2/0)	5 (4/1)	6 (5/1)	13 (3/10)	1 (1/0)	9 (8/1)	3 (3/0)	158 (110/48)
<b>Understanding Control Flow</b>																							
7. Which entities (functions, lines of code) are the most executed?	1 (1/0)	1 (1/0)	3 (3/0)	1 (1/0)	2 (2/0)	2 (2/0)		1 (1/0)		1 (1/0)	1 (1/0)			2 (2/0)									15 (15/0)
8. Where is this method called or referenced?	1 (0/1)	1 (1/0)	1 (1/0)		2 (2/0)		1 (1/0)		1 (0/1)							1 (1/0)			1 (1/0)				10 (8/2)
9. When during the execution is this method called?	2 (0/2)			1 (0/1)	1 (1/0)	1 (0/1)																	5 (1/4)
10. Which functions are called by this function?			2 (2/0)		2 (2/0)	4 (4/0)	1 (1/0)		2 (2/0)				1 (1/0)	1 (1/0)	1 (1/0)	1 (1/0)			1 (1/0)	4 (4/0)		3 (3/0)	1 (1/0)
11. How many times is this entity (function or line of code) executed?	2 (0/2)	2 (0/2)	2 (2/0)		8 (8/0)	1 (1/0)	2 (2/0)	2 (2/0)	2 (2/0)		2 (2/0)						1 (1/0)						26 (22/4)
12. How many recursive calls happen during this operation?						1 (1/0)											2 (1/1)						3 (2/1)
13. Which execution path is being taken in this case?		1 (0/1)	1 (1/0)		1 (1/0)	2 (2/0)	2 (1/1)	2 (1/1)	6 (5/1)		2 (1/1)	6 (4/2)	7 (7/0)	2 (2/0)	3 (2/1)			2 (2/0)			5 (2/3)	3 (3/0)	4 (4/0)
14. Under what circumstances is this method called or exception thrown?	1 (0/1)	1 (1/0)		1 (1/0)	3 (3/0)	2 (2/0)	2 (2/0)	1 (0/1)				2 (2/0)	1 (0/1)		4 (3/1)						2 (2/0)		20 (16/4)
15. In what order are these functions executed?	1 (1/0)				1 (1/0)	1 (1/0)						3 (2/1)											6 (5/1)
<i>Total in the category</i>	7 (1/6)	7 (4/3)	9 (9/0)	3 (2/1)	20 (20/0)	14 (13/1)	8 (7/1)	6 (4/2)	11 (9/2)	1 (1/0)	5 (4/1)	12 (9/3)	12 (11/1)	3 (3/0)	9 (7/2)	1 (1/0)	4 (3/1)	2 (2/0)	11 (8/3)	3 (3/0)	7 (7/0)	8 (7/1)	163 (135/28)
<b>Discovering Memory Usage in a Single Point of Time</b>																							
16. How much memory does this entity (function, instance, line of code) consume?	3 (3/0)		2 (1/1)		5 (4/1)	1 (1/0)	2 (2/0)	14 (14/0)	3 (3/0)	2 (2/0)	4 (4/0)	1 (0/1)	3 (1/2)					3 (2/1)	1 (1/0)		2 (2/0)		46 (40/6)
17. How much memory do these parts of the code consume together?	3 (2/1)	3 (1/2)	4 (4/0)		1 (0/1)	3 (1/2)	5 (2/3)	3 (1/2)	1 (1/0)			1 (0/1)	4 (2/2)	4 (1/3)	1 (0/1)	1 (0/1)	2 (3/0)	3 (1/4)	5 (1/0)	1 (1/0)	1 (1/0)		46 (23/23)
18. How much memory in total is being consumed?			3 (2/1)		2 (2/0)	1 (0/1)		2 (0/2)				1 (0/1)			1 (0/2)						1 (0/1)	1 (0/1)	14 (5/9)
19. Why do these parts of code consume this amount of memory?	1 (0/1)	2 (1/1)	4 (4/0)	3 (2/1)	4 (3/1)	1 (1/0)	1 (1/0)	5 (3/2)	6 (3/3)	1 (1/0)		2 (0/2)			1 (0/1)	1 (1/0)	1 (0/1)	1 (1/0)	1 (1/0)	17 (4/13)	3 (0/3)		55 (25/30)
20. How much is the maximum memory peak?					1 (1/0)			1 (1/0)				1 (0/1)											3 (1/2)
<i>Total in the category</i>	6 (5/1)	4 (1/3)	11 (8/3)	4 (4/0)	9 (6/3)	10 (7/3)	9 (5/4)	15 (15/0)	14 (8/6)	9 (6/3)	6 (5/1)	3 (1/2)	9 (3/6)	4 (1/3)	3 (1/2)	3 (1/2)	5 (1/4)	7 (6/1)	23 (6/17)	4 (1/3)	4 (3/1)	2 (0/2)	164 (94/70)
<b>Comparing and Contrasting Memory Usage</b>																							
21. How does memory usage evolve over time?			1 (1/0)			1 (0/1)		1 (1/0)	5 (3/2)				2 (0/2)		1 (0/1)		1 (0/1)		2 (0/2)		1 (0/1)		15 (5/10)
22. Which entities (functions, lines of code, instances) allocate most memory?	9 (6/3)	5 (3/2)	6 (4/2)	5 (4/1)	7 (2/5)	11 (5/6)	5 (1/4)	12 (8/4)	3 (0/3)	7 (4/3)	3 (1/2)	2 (1/1)	5 (4/1)	3 (0/3)	3 (1/2)	3 (2/5)	7 (4/2)	6 (4/10)	14 (3/1)	4 (3/4)	7 (1/2)	3 (1/2)	130 (62/68)
23. What will be the impact in memory consumption of this change?	1 (0/1)	1 (0/1)	3 (3/0)	1 (1/0)	2 (0/2)	1 (1/0)	3 (1/2)	1 (1/0)		1 (0/1)		2 (0/2)			1 (1/0)			2 (0/2)	8 (3/5)	3 (1/2)			30 (12/18)
24. What is the difference in memory consumption between these similar parts of the code (e.g., between sets of methods)?	1 (1/0)				1 (1/0)										1 (1/0)				1 (0/1)			4 (3/1)	
25. What are the differences in memory consumption between this point of time and that point of time?			1 (0/1)		2 (0/2)	1 (0/1)	1 (0/1)				1 (0/1)		1 (0/1)		2 (0/2)				10 (0/10)			1 (0/1)	
26. What is the difference in memory consumption between these code executions?		2 (1/1)	1 (0/1)			2 (1/1)			1 (0/1)	2 (0/2)			1 (0/1)								1 (0/1)		10 (3/7)
27. What part (function, line of code, instance) of this function consumes the most memory?	1 (1/0)	1 (1/0)				2 (2/0)	3 (3/0)	2 (2/0)		1 (1/0)	1 (1/0)	1 (1/0)	2 (2/0)	1 (1/0)	2 (2/0)	1 (1/0)			6 (6/0)	2 (2/0)	3 (3/0)	1 (1/0)	30 (30/0)
<i>Total in the category</i>	10 (7/3)	10 (6/4)	10 (5/5)	8 (7/1)	10 (3/7)	18 (7/11)	19 (4/5)	12 (13/6)	9 (7/5)	9 (4/5)	7 (2/4)	3 (2/1)	12 (5/7)	5 (2/3)	9 (4/5)	5 (3/2)	9 (3/6)	8 (4/4)	42 (13/29)	9 (6/3)	11 (6/5)	5 (2/3)	239 (115/124)
<b>Discovering Memory Events</b>																							
28. Where in the code are memory allocations made in this function?	6 (6/0)		2 (2/0)		1 (1/0)	1 (1/0)	1 (1/0)		4 (4/0)		7 (7/0)			1 (0/1)					2 (1/1)		1 (0/1)		26 (23/3)
29. Where are instances of this class created?									1 (1/0)				1 (1/0)										2 (2/0)
30. When are these instances garbage collected?						4 (3/1)			2 (1/1)				1 (1/0)								1 (1/0)		8 (6/2)
31. What data is being modified in this code?				1 (1/0)	2 (2/0)	1 (1/0)			1 (1/0)			1 (1/0)			1 (1/0)								7 (7/0)
32. How does this data structure look at runtime?						1 (1/0)			3 (2/1)														4 (3/1)
33. What objects are created?						2 (2/0)																	2 (2/0)
34. Where is this variable or data structure being accessed?						1 (1/0)						1 (1/0)											2 (2/0)
<i>Total in the category</i>	6 (6/0)	0 (0/0)	2 (2/0)	1 (1/0)	3 (3/0)	10 (9/1)	1 (1/0)	0 (0/0)	11 (9/2)	7 (0/0)	7 (7/0)	3 (3/0)	1 (1/0)	1 (0/1)	1 (1/0)	0 (0/0)	0 (0/0)	0 (0/0)	2 (1/1)	0 (0/0)	2 (1/1)	0 (0/0)	51 (45/6)
<b>Total</b>	41 (23/18)	30 (13/17)	38 (29/9)	25 (22/3)	50 (35/15)	61 (42/19)	34 (23/11)	48 (39/9)	59 (40/19)	22 (13/9)	25 (19/6)	32 (25/7)	32 (33/15)	16 (9/7)	30 (20/10)	11 (7/4)	23 (11/12)	23 (17/6)	91 (31/60)	17 (11/6)	33 (25/8)	18 (12/6)	775 (499/276)

*instance, type)?”, and “6. What is the behavior that these entities (functions, lines of code, instances) provide together?”*

**Finding 1.1:** Participants asked questions about understanding the aspects of source code for (i) searching relevant code and (ii) locating anomalies or opportunities for improvement.

#### 4.4.2 Understanding Control Flow

Questions related to understanding the control flow (*e.g.*, exploring the relationships between functions, identifying when or in which situations some functions are called) belong to this category. This category contains nine questions, and 163 question occurrences (21%) were asked during work sessions. The most frequent questions in this category are: “13. Which execution path is being taken in this case?”, “12. How many times is this entity (function or line of code) executed?”, and “10. Which functions are called by this function?”

Participants commonly raised these questions when (i) comprehending the execution of a particular code that impacted the memory usage (allocations, releases, accesses) and (ii) detecting the root cause of a memory issue.

**Comprehending code execution.** To better understand why particular code parts (specific entities or allocation hotspots) were executed, participants often investigated the circumstances that caused their execution and the relationships between functions/methods. For instance, P12 identified a function that consumed most memory and asked “*Under what circumstances is this `__new__` function called?*” P12 explored the relationships between functions and discovered that `__new__` function was called several times to generate multiple data frames that later are transformed into arrays using the numpy package.

**Detecting the root cause of an issue.** Programmers also centered on the control flow to detect the parts of code responsible for excessive or inefficient memory usage. For example, P19 located an allocation hotspot that threatened memory consumption and asked “*Which execution path is being taken to generate this amount of objects?*” to trace the root cause of the issue and analyze if the current code could be modified.

**Finding 1.2** Participants raised questions about understanding the control flow for (i) comprehending the execution of a particular code that impacted the memory usage and (ii) detecting the root cause of a memory issue.

#### 4.4.3 Discovering Memory Usage in a Single Point of Time

This category involves questions about discovering memory usage at a single point in time. We detected five questions and 164 question occurrences (21%) in this category. We also

noticed that the most popular questions are: “19. *Why do these parts of code consume this amount of memory?*”, “16. *How much memory does this entity (function, instance, line of code) consume?*”, and “17. *How much memory do these parts of the code consume together?*”

Participants asked these questions when they (i) investigated the memory usage of particular entities and (ii) analyzed memory usage at a specific time. Answering these questions helps programmers understand how some entities impact memory consumption.

***Exploring memory usage of entities.*** Participants usually asked for the memory used by certain entities during the application’s execution. For example, P8 asked “*How much memory do the functions in charge of plotting my figures consume?*” to verify and confirm that these functions used most of the memory. Also, participants asked “19. *Why do these parts of code consume this amount of memory?*” especially when they found (i) unexpected entities as allocation hotspots or (ii) code that consumed more or less memory than expected.

***Analyzing memory usage at a specific time.*** Some participants look for information about the total memory used at a particular time. For instance, P3 asked “*How much memory in total was consumed so far?*” to analyze the impact in memory usage of some functions during the execution. Also, some participants asked “20. *How much is the maximum memory peak?*” to detect the amount of allocated memory at the time when the memory usage was at its peak.

**Finding 1.3** Participants asked questions about discovering memory usage at a single point in time for examining the memory used by particular entities or on a specific time.

#### 4.4.4 Comparing and Contrasting Memory Usage

Questions about comparing and contrasting memory usage belong to this category. We identified seven questions and 239 question occurrences (31%) grouped in this category. Question “22. *Which entities (functions, lines of code, instances) allocate most memory?*” is the most frequently raised by participants.

Participants often raised questions from this category when they (i) detected allocation hotspots, (ii) investigated memory over time, and (iii) contrasting memory usage.

***Detecting allocation hotspots.*** To locate memory anomalies and opportunities for improvement, participants compared the memory consumed by entities to determine the allocation hotspots; thus, programmers asked “22. *Which entities (functions, lines of code, instances) allocate the most memory?*” and “27. *What part (function, line of code, instance) of this function consumes the most memory?*”

***Investigating memory over time.*** Some participants focused on how memory consumption varies during program execution. The latter helps programmers detect memory growth and potential leaks. Consequently, participants asked: “21. *How does memory usage evolve*



over time?”, and “25. What are the differences in memory consumption between this point of time and that point of time?”

**Contrasting memory usage.** Some participants tried to find opportunities to reduce memory usage by analyzing the memory used by parts of the code (“24. What is the difference in memory consumption between these similar parts of the code (e.g., between sets of methods)?”) or distinct code executions (“26. What is the difference in memory consumption between these code executions?”) As a result, programmers usually need to execute the code multiple times and compare the memory usage. Also, we noticed that when programmers propose a change, they often want to determine how it affects memory consumption and functionality (“23. What will be the impact on memory consumption of this change?”)

**Finding 1.4** Participants raised questions about comparing and contrasting memory usage when (i) detecting allocation hotspots, (ii) investigating memory over time, and (iii) contrasting memory usage.

#### 4.4.5 Discovering Memory Events

Questions about discovering and locating memory events are grouped in this category. A total of seven questions and 51 question occurrences (7%) were raised by participants and belong to this category. The most frequent question is “28. Where in the code are memory allocations made in this function?”

Participants asked these questions to detect an anomaly and optimization opportunity by exploring (i) memory allocations, (ii) memory accesses, and (iii) memory releases.

**Exploring memory allocations.** Some participants inspected the data allocated in memory and the code responsible for this action. Consequently, participants asked: (“28. Where in the code are memory allocations made in this function?”, “29. Where are instances of this class created?”, “32. How does this data structure look at runtime?”, “33. What objects are created?”)

**Exploring memory accesses.** Since the data allocated is used in different operations (read and write). Some participants explored how the data allocated in memory was used and why. As a result, participants asked: “31. What data is being modified in this code?” and “34. Where is this variable or data structure being accessed?”

**Exploring memory releases.** Some participants asked “30. When are these instances garbage collected?” to examine if memory occupied by data that is no longer needed was released.

**Finding 1.5** Participants asked questions about understanding memory events to detect an anomaly and optimization opportunity. They usually explored (i) memory allocations, (ii) memory accesses, and (iii) memory releases.

## 4.5 Results: How do Python Programmers Answer These Questions Using Vismep and Tracemalloc?

We detected that 665 question occurrences (86%) raised by participants that were answered using features from the selected memory profilers. Consequently, a total of 110 question occurrences (14%) asked by participants either (i) cannot be answered using the features of Vismep or Tracemalloc or (ii) practitioners are not able to benefit from memory profiler features (difficulties at interpreting or finding the required information).

To present detailed information about the tool usage and support, we analyzed the information required, the actions performed, and the features from Vismep and Tracemalloc that were employed to answer the questions. This section is organized around the five categories of questions.

### 4.5.1 Understanding Static Structure and Implementation

Participants responded to 150 question occurrences (95% of the occurrences in category) about understanding static structure and implementation during the work sessions.

Responding to question 1 “*Which entities (functions, lines of code, instances) are involved in the implementation of this behavior?*” is about finding code parts to a particular functionality. To answer this question using Vismep, participants usually searched for a specific entity based on its name or structural component (file or module). Participants often then check the entity’s code to ensure it is related to specific behavior and explore their relationships with other entities that could be involved in the same task. Consequently, participants often employed the *Call graph view* and the *Source code view*. For Tracemalloc, a participant manually inspected the source code of allocation hotspots based on the static information about the file and line number reported with *Display TOP* to verify if some hotspots were involved with the functionality that he considered problematic.

Questions 2 “*Which entities (functions, lines of code, class) belong to this file or module?*” and 3 “*Is there an entity named something like this in that unit (project, package, or class)?*” require finding entities based on their structural component (file or module they belong to) or name. To answer these questions with Vismep, participants often (i) manually inspect and search the name and file of entities in diverse views (*Call graph view*, *Scatter plot view*, *Sub call graph view*) using the popup windows, (ii) use the search mechanism to locate an entity with a particular name, and (iii) explore the *Source code view* of entities to ensure that they belong to a specific unit. When participants employed Tracemalloc, they answered question 2 by manually searching the names of files or modules in the textual reports (*Display TOP*, *Get traceback*) and exploring their source code.

Questions 4 “*What does the declaration or definition of this look like?*” and 5 “*What are the parts of this entity (function, instance, type)?*” involve inspecting the source code of entities. Participants responded to these questions by inspecting the source code of a given entity using *Source code view* from Vismep (see Figure 4.4). Also, participants explored the



## 4.5.2 Understanding Control Flow

A total of 137 question occurrences (84% of the occurrences in the category) were answered during sessions.

Questions 7 “Which entities (functions, lines of code) are the most executed?”, 11 “How many times is this entity (function or line of code) executed?” and 12 “How many recursive calls happen during this operation?” is about how entities were executed. To respond to these questions, participants used Vismep to examine the information about the entity’s execution with the popup windows or visual hints. For instance, participants often used *Scatter plot view* to investigate the position of entities since the X-axis in the chart represents the number of executions. Participants also selected *Call graph view* or *Sub call graph view* to inspect the height of nodes (proportional to the number of executions) and look for nodes with loops among their edges to identify recursion. Participants could not respond to questions 11 and 12 using Tracemalloc.

Answering questions 8 “Where is this method called or referenced?” and 10 “Which functions are called by this function?” require examining control flow information, specifically the relationship between functions. When participants used Vismep, they responded to these questions by exploring the relationships between nodes in *Call graph view* and *Sub call graph view* as shown in Figure 4.5. Sometimes they also inspected the code of certain functions/methods to indicate the line of code responsible for calling a function (*Source code view*). Participants could not answer question 8 employing Tracemalloc.

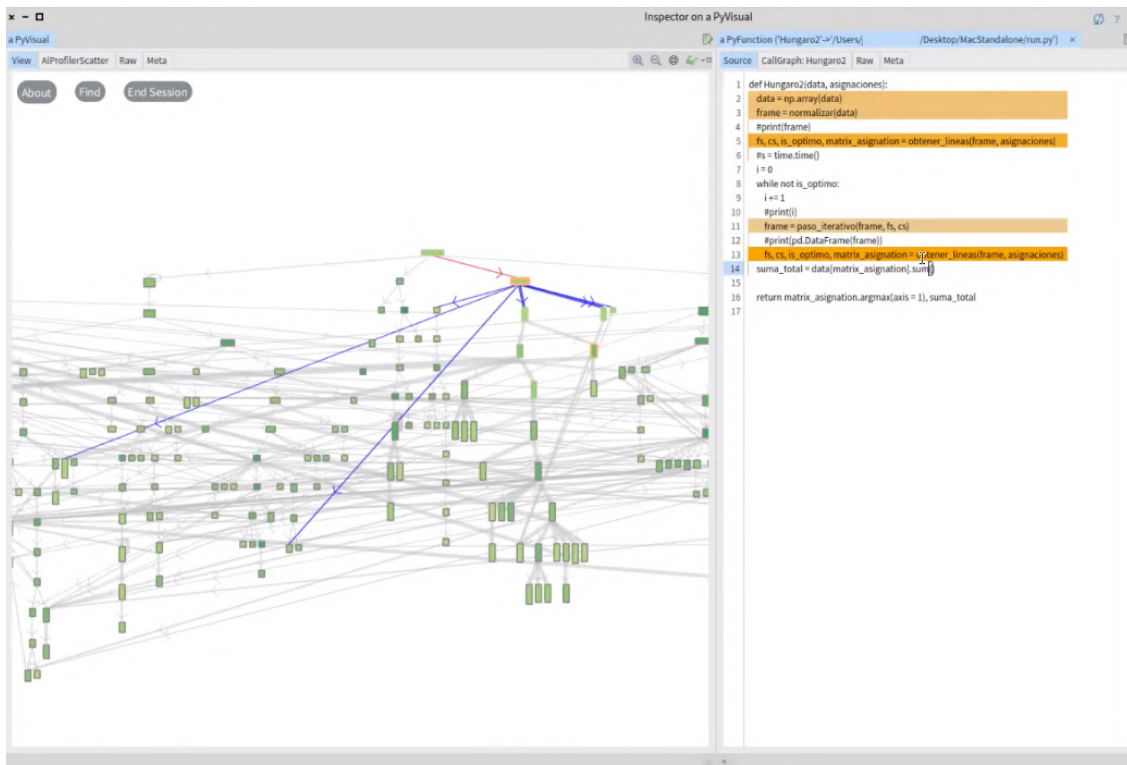


Figure 4.5: P9 explores the calling relationships between functions in *Call graph view* and inspects which functions are called in the code using the *Source code view*.

Questions 9 “*When during the execution is this method called?*”, 13 “*Which execution path is being taken in this case?*”, 14 “*Under what circumstances is this method called or exception thrown?*” and 15 “*In what order are these functions executed?*” consider understanding dynamic aspects of the control flow or data flow in a particular context. To answer questions 13, 14, and 15 using Vismep, participants investigated and navigated iteratively over the (i) code of functions (*Source code view*) and (ii) relationships between functions (*Call graph view*, *Sub call graph view*). When participants employed Tracemalloc, they answered these questions by (i) analyzing the chain of executions that lead to a particular memory allocation (*Get traceback*) and (ii) searching manually for the references to functions/methods and inspecting the respective code.

**Finding 2.2** Questions about understanding control flow usually consider exploring information about relationships between entities. In most cases, obtaining this information is relatively well supported by Vismep (*Call graph view* or *Sub call graph view* combined with other views) and Tracemalloc (*Get traceback*).

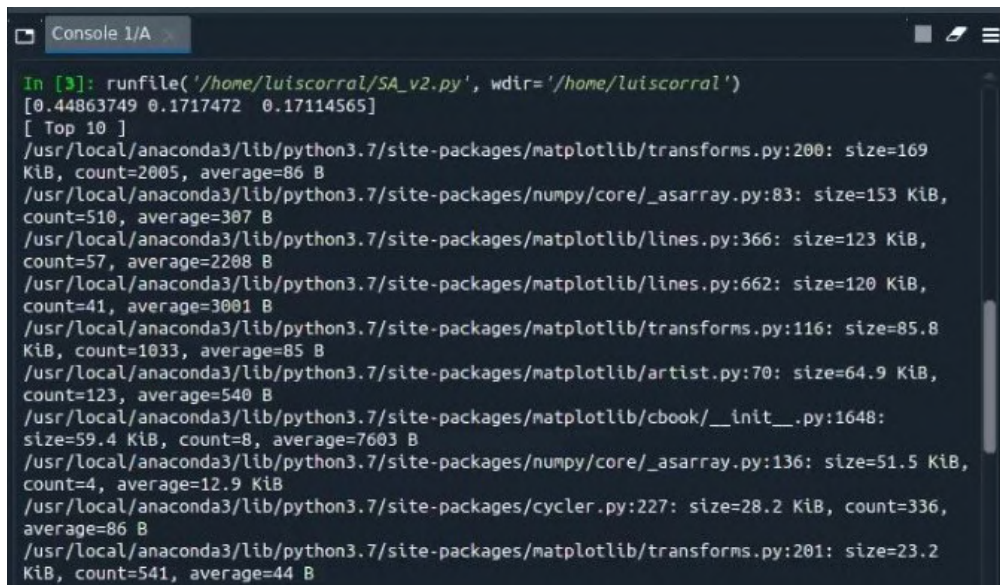
### 4.5.3 Discovering Memory Usage at a Single Point of Time

Participants responded to 144 question occurrences (88% of the occurrences in the category) about discovering memory usage at a point in time.

Responding to question 16 “*How much memory does this entity (function, instance, line of code) consume?*” and 17 “*How much memory do these parts of the code consume together?*” require exploring the memory usage of one or multiple entities altogether. When participants answered these questions with Vismep, they often investigated the information about memory usage of entities in the views (*Call graph view*, *Scatter plot view*, *Sub call graph view*). For Tracemalloc, participants used API calls (*Display TOP*, *Compute differences*) that report the memory usage of lines of code as shown in Figure 4.6. Besides, with both tools, participants performed mental operations with the information from each group entity to respond to question 17.

Questions 18 “*How much memory in total is being consumed?*” and 20 “*How much is the maximum memory peak?*” are about exploring the memory allocated at a point. Some participants answered question 18 with Vismep by locating the root function responsible for the program execution and inspecting its memory used in *Call graph view* and *Scatter plot view*. Participants could not respond to question 20 with Vismep. In the case of Tracemalloc, participants selected API calls to show the total memory usage (*Display TOP*, *Get traced memory*) and the memory peak (*Get traced memory*) in the textual reports.

Question 19 “*Why do these parts of code consume this amount of memory?*” considers dynamic and static aspects of the program to understand the reasons behind memory usage. To answer this question, participants usually focused on selected code parts and explored (i) the memory used by them, (ii) other code parts associated with their execution (control flow), and (iii) its source code to gain a better comprehension of program behavior. Consequently, when using Vismep, participants usually inspected the memory usage information (detailed in popup



```
Console 1/A
In [3]: runfile('/home/luiscorral/SA_v2.py', wdir='/home/luiscorral')
[0.44863749 0.1717472 0.17114565]
[ Top 10 ]
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/transforms.py:200: size=169 KiB, count=2005, average=86 B
/usr/local/anaconda3/lib/python3.7/site-packages/numpy/core/_asarray.py:83: size=153 KiB, count=510, average=307 B
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/lines.py:366: size=123 KiB, count=57, average=2208 B
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/lines.py:662: size=120 KiB, count=41, average=3001 B
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/transforms.py:116: size=85.8 KiB, count=1033, average=85 B
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/artist.py:70: size=64.9 KiB, count=123, average=540 B
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/cbook/__init__.py:1648: size=59.4 KiB, count=8, average=7603 B
/usr/local/anaconda3/lib/python3.7/site-packages/numpy/core/_asarray.py:136: size=51.5 KiB, count=4, average=12.9 KiB
/usr/local/anaconda3/lib/python3.7/site-packages/cycler.py:227: size=28.2 KiB, count=336, average=86 B
/usr/local/anaconda3/lib/python3.7/site-packages/matplotlib/transforms.py:201: size=23.2 KiB, count=541, average=44 B
```

Figure 4.6: P8 used the API calls from *Display TOP* to enlist the parts of code that allocated the most memory and focused on the amount of memory allocated by specific lines of code.

windows), the relationships between functions (*Call graph view* or *Sub call graph view*), and the code (*Source code view*). For Tracemalloc, some participants often reported the memory used by parts of code and inspected in depth its source code. Other participants reported the changes in memory usage after and before executing these parts of code (*Compute differences* along with the chain of code execution that caused the memory allocations associated with them (*Get traceback*).

**Finding 2.3** Questions about discovering memory usage at a point typically involve examining information about the memory allocated by entities or at a point. Some questions also require exploring code and relationships between entities. In general, answering questions in this category is well supported by Vismep (*Call graph view* and other views combined) and Tracemalloc (includes memory used by entities in various reports and at a point with *Get traced memory*).

#### 4.5.4 Comparing and Contrasting Memory Usage

A total of 199 question occurrences (83% of the occurrences in the category) were answered by participants.

Responding to question 21 “*How does memory usage evolve over time?*” and 25 “*What are the differences in memory consumption between this point of time and that point of time?*” considers understanding changes in memory usage over time. When participants used Tracemalloc, they answered question 21 by answering multiple occurrences of question 25. They often report if the memory increased or decreased after and before executing a function using the API calls from *Compute differences* (see Figure 4.7). Consequently, some



participants obtained information about the changes in memory usage over time by inspecting several reports from *Compute differences*. Some participants did not respond to question 21 using Vismep, and question 25 was not asked when Vismep was used.

```

Select Windows PowerShell
C:\Python38\lib\site-packages\nbt\nbt.py:488: size=17.3 KiB (+17.3 KiB), count=327 (+327), average=54 B
C:\Python38\lib\site-packages\nbt\nbt.py:42: size=40.9 KiB (+40.9 KiB), count=804 (+804), average=52 B
C:\Python38\lib\site-packages\nbt\nbt.py:182: size=14.4 KiB (+14.4 KiB), count=14 (+14), average=1052 B
C:\Python38\lib\site-packages\nbt\nbt.py:353: size=22.7 KiB (+22.7 KiB), count=358 (+358), average=65 B
C:\Python38\lib\site-packages\nbt\nbt.py:10: size=13.4 KiB (+13.4 KiB), count=324 (+324), average=42 B
C:\Python38\lib\site-packages\nbt\nbt.py:488: size=15.7 KiB (+15.7 KiB), count=292 (+292), average=55 B
C:\Python38\lib\site-packages\nbt\nbt.py:245: size=8676 B (+8676 B), count=14 (+14), average=620 B
C:\Python38\lib\site-packages\nbt\nbt.py:10: size=13.6 KiB (+13.6 KiB), count=327 (+327), average=43 B
C:\Python38\lib\site-packages\nbt\nbt.py:404: size=7024 B (+7024 B), count=131 (+131), average=54 B
C:\Python38\lib\site-packages\nbt\nbt.py:245: size=8672 B (+8672 B), count=24 (+24), average=619 B
C:\Python38\lib\site-packages\nbt\nbt.py:473: size=6720 B (+6720 B), count=120 (+120), average=56 B
C:\Python38\lib\site-packages\nbt\nbt.py:182: size=8420 B (+8420 B), count=8 (+8), average=1052 B
C:\Python38\lib\site-packages\nbt\nbt.py:492: size=4872 B (+4872 B), count=114 (+114), average=43 B
C:\Python38\lib\site-packages\nbt\nbt.py:404: size=7016 B (+7016 B), count=132 (+132), average=53 B
Comparison number: 2
C:\Python38\lib\site-packages\nbt\nbt.py:473: size=5936 B (+5936 B), count=106 (+106), average=56 B
C:\Python38\lib\tracemalloc.py:532: size=768 B (+768 B), count=15 (+9), average=51 B
C:\Python38\lib\site-packages\nbt\nbt.py:492: size=4360 B (+4360 B), count=99 (+99), average=44 B
C:\Users\vepikadmin\Desktop\Perflador\VMCStructureLeamer\main.py:132: size=424 B (+424 B), count=1 (+1), average=424 B
Comparison number: 2
C:\Users\vepikadmin\Desktop\Perflador\VMCStructureLeamer\main.py:128: size=0 B (-424 B), count=0 (-1)
C:\Python38\lib\tracemalloc.py:532: size=768 B (+768 B), count=15 (+9), average=51 B
C:\Python38\lib\site-packages\nbt\nbt.py:105: size=1020 B (-240 B), count=36 (-5), average=28 B
C:\Users\vepikadmin\Desktop\Perflador\VMCStructureLeamer\main.py:132: size=424 B (+424 B), count=1 (+1), average=424 B
C:\Python38\lib\tracemalloc.py:397: size=800 B (+200 B), count=9 (+4), average=89 B
C:\Users\vepikadmin\Desktop\Perflador\VMCStructureLeamer\main.py:128: size=0 B (-424 B), count=0 (-1)
C:\Python38\lib\tracemalloc.py:534: size=624 B (+104 B), count=5 (+2), average=125 B
C:\Python38\lib\site-packages\nbt\nbt.py:105: size=992 B (-240 B), count=35 (-5), average=28 B
C:\Python38\lib\site-packages\nbt\nbt.py:42: size=44.4 KiB (-64 B), count=873 (-1), average=52 B
C:\Python38\lib\tracemalloc.py:397: size=800 B (+200 B), count=9 (+4), average=89 B
C:\Python38\lib\site-packages\nbt\nbt.py:97: size=424 B (-56 B), count=1 (-1), average=424 B
C:\Python38\lib\tracemalloc.py:534: size=624 B (-104 B), count=5 (+2), average=125 B
C:\Python38\lib\site-packages\nbt\nbt.py:81: size=32 B (-48 B), count=2 (-1), average=276 B
C:\Python38\lib\site-packages\nbt\nbt.py:42: size=40.8 KiB (-64 B), count=803 (-1), average=52 B
C:\Python38\lib\site-packages\nbt\nbt.py:481: size=520 B (-48 B), count=2 (-1), average=260 B
C:\Python38\lib\site-packages\nbt\nbt.py:97: size=424 B (-56 B), count=1 (-1), average=424 B
  
```

Figure 4.7: P19 used the API calls from *Compute differences* to understand how the memory is allocated and released when a selected function is executed multiple times. The report shows the lines responsible for changing memory usage each time the function is executed.

Questions 22 “Which entities (functions, lines of code, instances) allocate most memory?”, 24 “What is the difference in memory consumption between these similar parts of the code (e.g., between sets of methods)?” and 27 “What part (function, line of code, instance) of this function consumes the most memory?” is about comparing the memory allocated by entities and locating the code responsible for these allocations. To answer questions 22 and 24, participants center on locating allocation hotspots. For these cases, participants often explored and manually compared the visual cues of elements in diverse views of Vismep. For example, to identify allocation sites between functions, they searched the widest nodes in *Call graph view* or the points located at the top in *Scatter plot view* since the width and the position in Y-axis indicate the memory usage. To identify code lines or instances that allocate most memory in a function, participants inspected the *Source code view* and located the lines with a darker background. For *Tracemalloc*, participants responded to these questions by reporting with *Display TOP* the allocation hotspots along with the file and the line of number. In the case of question 24, participants compared the memory usage of code parts utilizing information from background color in *Source code view* using Vismep or checking changes using *Compute differences* with *Tracemalloc*.

Answering questions 23 “What will be the impact in memory consumption of this change?” and 26 “What is the difference in memory consumption between these code executions?” require to detect changes in memory usage between versions or executions. To respond to these questions using *Tracemalloc*, participants usually reported the memory usage during program execution through various features (*Display TOP*, *Get traced memory*) for each version or execution. Then, participants manually compared the information from these reports to locate changes in memory usage based on a change or a different input. Participants could not answer these questions with Vismep.

**Finding 2.4** Some questions about comparing memory usage consider exploring memory usage over time. Answering these questions is well supported by Tracemalloc (*Compute differences*). The remaining questions in this category commonly involve comparing memory consumed by entities, versions, and code executions. Some questions also require locating the code responsible for allocations. In most cases, obtaining this information is relatively well supported by Vismep (*Call graph view, Source code view* and combined views) and Tracemalloc (*Display TOP* and *Compute differences*).

#### 4.5.5 Discovering Memory Events

A total of 35 question occurrences (69% of the occurrences in the category) about discovering memory events were answered by participants.

Questions 28 “*Where in the code are memory allocations made in this function?*” is about locating the code responsible for any memory allocation inside a function. Participants answered question 28 by detecting code lines with a colored background in *Source code view* with Vismep (see Figure 4.8) and inspecting the code lines that allocate memory in reports (*Display TOP*, *Compute differences*) of Tracemalloc.

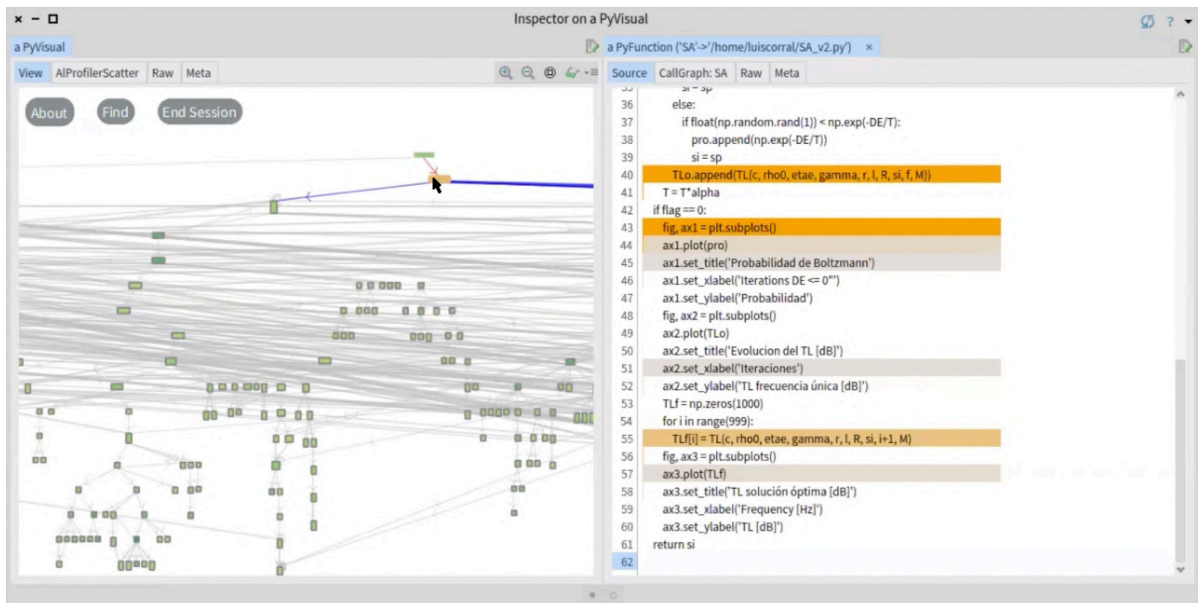


Figure 4.8: P8 inspects the code of SA function using the *Source code view* to identify the memory allocations made in the function by focusing on the highlighted lines.

Answering questions 29 “*Where are instances of this class created?*”, 30 “*When are these instances garbage collected?*” and 33 “*What objects are created?*” consider understanding information about the creation and release from the memory of instances. Participants could not answer question 29 using Vismep, and occurrences from this question did not arise when participants used Tracemalloc. To answer question 30, a participant detected and inspected the points associated with certain instances and when the memory decreased with



*Compute differences* from Tracemalloc. Participants could not respond to questions 30 and 33 using Vismep, and no occurrence from question 33 was asked by participants when using Tracemalloc.

Question 31 “*What data is being modified in this code?*”, 32 “*How does this data structure look at runtime?*” and 34 “*Where is this variable or data structure being accessed?*” are about exploring the state of a particular data structure or locating the code responsible for accessing it. To answer questions 31 and 34 with Vismep, participants often examined the source code associated with a particular data and explored their relationships with other entities using *Source code view* and *Call graph view*. Occurrences from questions 31 and 34 were not asked when participants used Tracemalloc. In addition, participants were unable to answer question 32 with either tool.

**Finding 2.5** Questions about discovering memory events generally involve locating code responsible for memory events (allocations, releases, accesses). Answering questions around allocations is partially supported by Vismep (*Source code view*, combined views) and Tracemalloc (*Display TOP*, *Compute differences*).

#### 4.5.6 Unanswered questions

We detected that 110 question occurrences (14%) asked during work sessions were not answered. We explored the difficulties and attempts that participants made to try to answer the questions from the five categories.

**Understanding source code.** Participants had difficulties answering questions 2 “*Which entities (functions, lines of code, class) belong to this file or module?*” and 3 “*Is there an entity named something like this in that unit (project, package, or class)?*” To detect entities based on their structural component (file, module) or name, participants manually inspected many entities to obtain the information required. The latter caused mental fatigue because there was no automatic way to do this operation. Additionally, when some participants respond to question 6 “*What is the behavior that these entities (functions, lines of code, instances) provide together?*”, they inspected relationships between several entities to better understand the group’s functionality, but they mentioned that this operation demands considerable mental effort.

**Finding 2.6** We detected that tool support for answering questions about source code could be limited when searching entities.

**Understanding control flow.** Some participants could not answer questions 9 “*When during the execution is this method called?*”, 13 “*Which execution path is being taken in this case?*”, 14 “*Under what circumstances is this method called or exception thrown?*” and 15 “*In what order are these functions executed?*” The latter occurs because participants struggle to get control or data flow information in a particular context. For example, participants failed to answer question 9 “*When during the execution is this method called?*” using Vismep and

Tracemalloc since it involves the context of a particular point, and these tools do not extract context information. Some participants did not respond to questions 8 “*Where is this method called or referenced?*”, 11 “*How many times is this entity (function or line of code) executed?*” and 12 “*How many recursive calls happen during this operation?*” using Tracemalloc because it lacks features to extract this information.

**Finding 2.7** Some participants considered the reported information insufficient due to (i) the lack of context in both tools and (ii) the level of support in analyzing relationships between functions and entity execution for Tracemalloc.

**Discovering memory usage in a point.** Some participants asked questions 16 “*How much memory does this entity (function, instance, line of code) consume?*” and 17 “*How much memory do these parts of the code consume together?*” at a fine-grained level (line of code, instance). However, they could not respond to these questions due to the lack of explicit information for these cases in Vismep and Tracemalloc. The participants that asked about memory used by lines of code in Vismep explored the *Source code view* and only found that the visual hints of background code were the unique information about memory usage at code line. Another example is when P2 did not find any suitable features in Tracemalloc to obtain the information required to answer the question “*How much memory does this variable consume?*”. Furthermore, one participant could not answer question 20 “*How much is the maximum memory peak?*” with Vismep since the profiler did not explicitly show the information. Finally, we noticed that participants had challenges with question 19 “*Why do these parts of code consume this amount of memory?*” since the navigation between dynamic information (memory usage, program execution) and static information (source code, structural component) was not always connected adequately.

**Finding 2.8** We detected that participants require more support in (i) understanding memory usage at a fine-grained level (line of code, instance) and (ii) connecting static and dynamic information.

**Comparing and contrasting memory usage.** Some participants did not answer questions 22 “*Which entities (functions, lines of code, instances) allocate most memory?*”, and 24 “*What is the difference in memory consumption between these similar parts of the code (e.g., between sets of methods)?*” using Vismep, since it does not explicitly present information about memory consumed by code lines. Participants often had difficulties manually comparing the information about memory usage displayed by multiple reports using Vismep and Tracemalloc. The latter is an action frequently performed to answer questions 23 “*What will be the impact in memory consumption of this change?*” and 26 “*What is the difference in memory consumption between these code executions?*” In addition, participants did not respond to question 21 “*How does memory usage evolve over time?*” with Vismep because it lacks adequate support in analyzing memory usage over time. Besides, participants also faced difficulties using Tracemalloc to answer this question. In some cases, they manually compared several reports and gave up due to high mental demand.

**Finding 2.9** We noticed that participants need more support in (i) understanding memory usage at a fine-grained level (line of code, instance), (ii) analyzing memory usage over time using Vismep and (iii) comparing dynamic information at different levels (*e.g.*, between executions and versions).

**Discovering memory events.** Participants failed to answer question 29 “*Where are instances of this class created?*” with Vismep, they tried identifying functions associated with creating specific instances and exploring their relationships to locate the code responsible for calling them. However, performing this search was tedious and expensive; as a result, participants concluded that the information was insufficient to answer this question directly. Most participants also mentioned that Vismep and Tracemalloc do not provide the required information to answer questions 30 “*When are these instances garbage collected?*” and 33 “*What objects are created?*” To answer question 32 “*How does this data structure look at runtime?*”, participants considered that there was no feature of tools that could be useful. Furthermore, one participant mentioned that Vismep does not provide enough information to answer question 34 “*Where is this variable or data structure being accessed?*”

**Finding 2.10** Participants often struggle when obtaining information about the (i) creation or release of instances and (ii) the state of data structures during program execution.

#### 4.5.7 Suggestions

We collected suggestions and observations from participants’ answers to the post-study questionnaire. In the following, we detail the information gathered for Vismep:

- *Entities from a particular file/module.* Most participants were surprised by the impact of external libraries on the complexity of their application. Due to this, the analysis of memory consumption was complex since some participants focused on exploring the memory usage of specific entities that do not belong to external libraries. Six participants explicitly suggested adding the option to filter or exclude functions based on their structural components. Additionally, they suggested adding information about the percentage of memory usage per structural component (*e.g.*, external libraries).
- *Fine-grained memory usage.* Some participants asked about memory used by variables or lines of code. Although Vismep provides visual hints with the code line background in *Source code view*, it does not present explicit information on how much memory a line of code or a variable consumes. Two participants explicitly recommended adding this information.
- *Metric selection.* A number of participants had difficulty comparing entities based on the default visual mapping. Therefore, two participants recommended adding an option for the user to select the mapping between metrics (*e.g.*, memory usage, number of executions) and visual features (*e.g.*, width, height, color).

- *Vismep performance.* Some participants highlighted the importance of improving the profiler’s performance since, in some cases, the data extraction took a long time compared to the execution time of the application to be analyzed.

The participants also provide some observations related to Tracemalloc:

- *Connecting dynamic with static information.* Most participants had difficulties exploring the dynamic information displayed in textual reports and inspecting the associated source code. For example, participants struggle when analyzing relationships between functions and entity execution.
- *Selecting and using features.* Some participants struggled to obtain the necessary information using Tracemalloc. Phrases like “*it is tiring having to add lines of code in specific places*” or “*I was frustrated using Tracemalloc, I could not get the level of detail I wanted in the information*” were frequent during work sessions. In addition, twelve participants dealt with errors when running the modified code to get the necessary information.

## 4.6 Discussion

We discuss our findings and contrast them with other prior work as necessary.

### 4.6.1 Questions asked by participants

We found three categories centered on memory usage: (i) discovering memory consumption at a single point in time, (ii) comparing and contrasting memory consumption, and (iii) discovering memory events. Although these findings may not be surprising since they are involved with primary activities (*e.g.*, performance, inspection of memory anomalies) mentioned in previous work [76, 156], we provide empirical evidence about their relevance.

We also detected questions about understanding the source code and the control flow. We included these questions because participants asked them to answer questions from the rest of the categories. To illustrate, after locating allocation hotspots, several participants asked “*Why do these parts of code consume this amount of memory?*” Then, to answer this question, participants inspected the source code of the particular allocation hotspot (“*What does the definition of this look like?*”) and explored the control flow to understand why that code is executed (“*Under what circumstances is this method called?*”). Our results show that programmers require support for software comprehension when addressing a memory issue or determining the root cause of a potential failure.

***Questions of other studies.*** We reported that participants asked questions that were and were not reported in studies on developer information needs. For example, most questions about discovering memory usage at a single point in time and comparing and contrasting

memory usage were not identified by any other study. This situation may occur due to the lack of information about the needs of programmers during memory consumption analysis [23]. Consequently, our study provides valuable information to guide researchers in designing tools that adequately support programmers in this context.

Furthermore, most questions we report about understanding source code and control flow were informed by studies about programming change tasks [72, 73, 133, 134]. In addition, some questions asked for discovering memory events were classified as questions related to data flow or performance in previous studies [73, 76]. These situations illustrate that programmers asked common questions about software comprehension, whether to perform programming change tasks, face memory issues or determine the cause of a failure.

**Questions frequency.** We divided our participants into two groups, G1 and G2. Participants from G1 first used Vismep (FP) and then Tracemalloc (SP), while in G2, the order of tools was reversed. We observed that participants from G1 usually ask more questions than participants from G2. We also noticed that questions about discovering memory events were asked more often in G1 than in G2. One explanation for this difference is the applications selected by participants and how they address issues. For example, some participants (P6, P9) from G1 chose applications with memory anomalies (leaks, bloats) and centered on exploring the memory allocations and releases when addressing these issues.

We also detected that questions from the first, second, and last categories arose more frequently when participants used Vismep. One reason for this difference is that programmers considered that Vismep gives more support to answer these questions than Tracemalloc; thus, more questions arose. In addition, questions from the third and fourth categories were asked more often during the first phase of both groups. One contributing factor may be the learning effect in the study. In other words, some participants asked questions about the program’s memory usage during the second phase considering the knowledge acquired from the first phase. For instance, some participants could avoid asking about the memory used by some entities they know are not a threat to memory usage based on analysis from the first phase.

**Learning effect.** Participants analyzed the same application with Vismep and Tracemalloc. The order for using each tool was defined based on the participant’s group (G1 and G2). Each participant decided when to end up the first phase to start the second phase and use another tool. Therefore, we did not force participants to switch from one tool to another. In the second phase, participants generally (i) located or confirmed the information or assumptions they had in the first phase, (ii) used the knowledge from the first phase to analyze the code further, and (iii) detected new information. We also noted that participants asked more or fewer questions between phases due to the tool’s limitations or because they had prior knowledge of the first phase. To illustrate, participants asked more questions about memory evolution over time using Tracemalloc due to the support that this tool offers. In addition, as mentioned before, we discussed that the frequency of questions might vary due to different reasons (*e.g.*, tool usage, application selected). Note that our goal is not to compare Vismep and Tracemalloc or rate the questions for relevance based on the frequency.

Furthermore, we increased the diversity and range of the questions asked by practitioners due to dividing participants into two groups balanced based on their study field and observing how they analyzed the same application with both tools in a different order. We also found

that programmers analyze memory usage and address memory issues using a wide range of features that are not necessarily supported by a single tool.

**Study field effect.** We examined if the questions asked may vary based on the participant’s study field. We analyzed the differences in questions between two groups of participants: (i) group C, which contains participants in the Computer Science field, and (ii) group N, which contains the rest of the participants. As a result, group N participants often asked more questions about understanding source code and control flow. In addition, group C participants usually asked more questions about discovering memory usage at a point in time and comparing memory usage. These differences could be that participants from fields distinct to Computer Science may need to explore several entities at different levels to understand the program’s behavior and structure.

**Self-assessment expertise effect.** We inspected if the questions asked may vary based on the participant’s expertise in Python. We analyzed the differences in questions between two groups of participants: (i) group E, which contains participants with self-assessment expertise above 3.2 (average), and (ii) group N, which contains the rest of the participants. As a result, group N participants usually asked more questions about understanding source code, control flow, and discovering memory events. Group E participants often asked more questions about discovering memory usage at a point in time and comparing memory usage. A factor that may cause these differences could be that participants considered themselves with expertise above regular in Python could obtain information to modify their code, optimize memory usage and perform fewer operations to comprehend the program’s functionality.

**Gender effect.** Some studies [15, 16] have shown differences between males and females when debugging. Our study involved twenty-two participants, of which six were female. We detected that both male and female programmers ask questions from the five categories. We also found that female programmers usually ask more questions about discovering memory events, while in the other categories, the question frequencies are often higher for male programmers. However, we could not ensure that these situations are due to gender since our study presents (i) a small sample, which is not balanced between males and females, and (ii) variations in sessions (*e.g.*, applications selected). Nonetheless, studies of the gender-related issues within tasks related to memory consumption analysis would significantly extend this work.

Furthermore, any observations about the occurrences of the questions should be treated carefully as the sessions from which the data is extracted varied in diverse ways (*e.g.*, different applications, presence of memory issues). Notice that we cannot conclude that the reason for differences among questions asked is due to the background of participants (*e.g.*, study field, experience in Python) due to the variations in sessions.

## 4.6.2 Answering questions

We observed that Vismep and Tracemalloc could support programmers in answering most questions inferred in this study. However, Section 4.5.6 details that these tools could present limitations in answering some questions. Consequently, we believe programmers need more

comprehensive support in (i) comparing and piecing information together, (ii) connecting dynamic information with source code. Besides, programmers require extracting information about dynamic aspects at different levels (*e.g.*, control flow in context, memory usage by code lines and instances).

***Support of other tools.*** We believe that most libraries/tools in Python might support most questions inferred in our study due to the reported information and the activities they claim to support. For example, some options [4, 20] could help answer most questions in comparing and contrasting memory usage since they extract information about (i) the allocation hotspots and (ii) memory usage over time (explicit information about time according to documentation). In addition, other options [1, 2, 6] may provide adequate support for answering some questions about memory events, specifically creation and release of specific instances. Furthermore, we consider that some options could not adequately support answering questions about source code and control flow because they do not collect information from these aspects, or the way of connecting dynamic information and source code may be insufficient.

Nonetheless, further research must be conducted with other tools to confirm if programmers could use them to answer these questions.

***Learning effect.*** We defined that a question is answered only if the participant demonstrates and explains how she/he employs the corresponding tool to respond to the question and mentions the answer. Vismep and Tracemalloc differed in the information reported and how information is provided and navigated. Consequently, programmers used different strategies to manage both tools and obtain certain information. For example, participants compared the visual hints of elements in the *Call graph view* and *Source code view* to identify allocation hot spots with Vismep. On the other hand, participants used the *Display TOP* from Tracemalloc to obtain the same information.

***Study field effect.*** We observe that participants, regardless of their study field and tool's usage, usually answered a similar proportion of questions for all categories except understanding control flow. Computer science participants tend to answer more questions about control flow. The latter may be because these participants may be more accustomed to performing operations to explore information on this aspect.

***Self-assessment expertise effect.*** We analyzed the differences in questions between two groups of participants: (i) group E, which contains participants with self-assessment expertise above 3.2 (average), and (ii) group N, which contains the rest of the participants. Participants from both groups, regardless of the tool's use, often answered a similar proportion of questions for all categories except discovering memory events. Participants from group E usually responded to more questions about discovering memory events. One factor may be that experienced participants could extract information by using different features to obtain information that was not explicitly displayed (*e.g.*, allocations, releases).

***Gender effect.*** We noticed that participants, regardless of gender, often answered a similar proportion of questions for all categories except discovering memory usage at a single point and discovering memory events. Female programmers tend to answer more questions about discovering memory usage at a single point and discovering memory events than male

programmers. This situation could occur because females were found efficient users of tinkering in previous studies [15, 16].

## 4.7 Threats to validity

Our study and results are subject to validity threats [162]. To carefully identify possible threats and analyze how their impact may be mitigated, we decided the following:

**Conclusion validity.** Our conclusions are founded on an exploratory study involving programmers analyzing the memory usage of their Python applications using memory profilers. However, our conclusions are based on observing only twenty-two participants, a relatively low sample. We try to reduce this threat by selecting participants with diverse backgrounds (*e.g.*, study fields, experience in Python programming). Although we had no indication that increasing the number of participants may invalidate our result, the frequency of questions in our results may be affected.

**Internal validity.** The student supervised by the thesis author performed the data collection and transcription of each work session. Then, the thesis author checked the generated spreadsheets based on the video recordings and event logs to minimize inconsistencies. Furthermore, the thesis author conducted the steps to identify the questions asked by the participants. Identifying concrete questions based on user behavior may be inaccurate, as participants did not explicitly verbalize their thoughts. Finally, the process of defining general questions may suffer from uncertainty. For instance, it can be challenging to distinguish question 23 "What will be the impact in memory consumption change?" from question 26 "What is the difference in memory consumption between these code executions?". To minimize inaccuracy in the inference process, the thesis author contrasted different scenarios and events related to the same questions and checked if the inferred questions were consistent with the information from spreadsheets. All the authors held two meetings to discuss any discrepancy in the inferred questions' consistency.

Regarding the classification scheme, the thesis author conducted a thematic analysis to organize the inferred questions based on the information needed and the programmer's behavior to answer a question. The codes and themes generated vary depending on the coder's experience, level of abstraction, and point of view. To mitigate this threat, two supervisors of this thesis checked the consistency of the process by examining the description of themes with the associated data. We conducted two meetings to discuss and resolve the disagreements among generated codes and themes. Additionally, two reviewers independently categorized the inferred questions using the classification scheme, and a measure of agreement between reviewers was calculated to validate the reliability.

**Construct validity.** We focus on understanding how programmers analyze the memory usage of Python programs using memory profiler tools. Therefore, we voluntarily centered on the Python programming language. For the study, participants chose software applications with which they were familiar to analyze them during the sessions. Furthermore, data from each work session was carefully examined and collected using records, logs, and observation.



**External validity.** The selected memory profiler tools and the individual differences among participants influence the programmers' questions and how they answer those. Consequently, our data could be different given a completely distinct set of memory profiler tools or participants. The latter must be considered when interpreting or generalizing the results. We mitigated this threat by selecting programmers with different backgrounds and experience levels. In addition, we opted for memory profiler tools that provide diverse information and report presentations commonly offered by other tools for the memory usage analysis. We also discussed and analyzed the tool support for answering questions considering the features of the tools. As a result, we noticed that many of the inferred questions were independent of whether they could be answered with the tools or not. Furthermore, no new questions were detected in the last work sessions. The latter suggests that some of our results will likely generalize to other Python tools. However, a follow-up study in which participants were asked to work with another set of memory profiler tools would help demonstrate that one or more questions have more or no support than noted in our analysis.

Our results also need to be interpreted relative to the programming language and the open questions used in the study. We focused on understanding the impact of memory profiler tools on supporting programmers during memory consumption analysis for Python applications. The participants selected Python applications with which they were familiar and answered open questions to ensure that participants had a goal they cared about and looked for the information they considered valuable. We do not diversify our study by covering other programming languages due to the difficulty of conducting such studies on a large scale. For example, data collection and transcription are expensive as it takes about one day to complete this process per work session. Furthermore, our study does not present tasks more specific and detailed (*e.g.*, identify five allocation hotspots) since our goal is to gather questions that programmers consider relevant. Consequently, diversifying the study in these aspects would be valuable for prioritizing future research and efforts around building tools to support the needs in particular situations.

As mentioned above, the sessions in our study varied along several dimensions, and we have not thoroughly analyzed how the questions asked and the answering behavior varied along those dimensions. Although we discussed in Section 4.6 the differences between question frequencies and the questions asked and answers by participants with diverse backgrounds, this information is insufficient to conclude that the study field or the programming experience affects the questions asked and how the tools are used. Obtaining precise information about the latter would require a study set up with more controls on the dimensions along which the sessions are allowed to vary.

## 4.8 Summary

We conducted an observational study to provide (i) an empirically-based set of questions that programmers ask during memory usage analysis and (ii) a report about how programmers those questions with Vismep and Tracemalloc. In our exploratory study, we observed 22 programmers analyzing the memory consumption of Python applications with which they were familiar using these two tools.

**Asking questions.** We found 34 different questions raised by participants and organized them into five categories based on the information needed and the programmer’s behavior: (i) understanding source code, (ii) understanding control flow, (iii) discovering the memory usage at a single point of time, (iv) comparing and contrasting memory usage, and (v) discovering memory events.

**Answering questions.** We noticed that answering most questions of our study is generally well supported by Vismep and Tracemalloc. However, participants often had difficulties with these tools in some activities (i) searching entities, (ii) understanding control flow in a particular context, (iii) understanding memory usage at a fine-grained level (line of code, instance), (iv) connecting static and dynamic information, (v) comparing dynamic information at different levels (*e.g.*, between executions and versions) and (vi) exploring creation, release and the state of data structures. Consequently, we consider that programmers require more comprehensive support in (i) comparing and piecing information together and (ii) connecting dynamic information with source code.

We also discussed that some prior work [22, 154, 156] and other tools may provide support to answer the questions inferred in our study due to the information collected and the tasks they claim to support. However, a follow-up study will help recognize the degree of support that may provide other tools for programmers to answer these questions.

The significant implication is that there is still much to learn about how programmers analyze memory usage. Multiple studies beyond this are needed to thoroughly understand how programmers perform these activities in different contexts. Those insights can be leveraged to create new tools or improve current tools to support programmers in their development environments.

# Chapter 5

## Conclusions

This chapter summarizes the dissertation contributions, discusses our studies, and points to possible future works.

### 5.1 Dissertation Contributions

This dissertation makes a number of contributions which are summarized below considering the addressed research questions.

***Literature review.*** We reported a systematic literature review of published works centered on software visualizations for analyzing the memory consumption of programs in Chapter 2. We have systematically selected 46 articles and categorized them based on the tasks supported, data abstracted, visual representations, evaluations conducted, and prototype availability. As a result, we introduce a taxonomy based on these five dimensions to identify the main challenges of visualizing memory consumption and opportunities for improvement. This study answers the following five research questions:

- “Which tasks are supported by the software visualizations to help users with the analysis of memory consumption?” We used thematic analysis to generate a classification scheme since an empirical catalog of tasks that shows the programmers’ needs or requirements during memory usage analysis and memory issue repairing is missing. As a result, we classified the articles based on (i) focus point analysis and (ii) memory issue detection. According to the classification in focus point analysis, most software visualizations focus on supporting programmers in data structure and general purposes involved with memory usage analysis, such as heap analysis. Furthermore, the research on visualizations to support memory regression or cache performance analysis is limited. We also noticed that around 47.82% of the visualizations support memory issue detection. Consequently, memory issue identification is not fully explored yet, leaving an open opportunity.
- “What aspects of the software are abstracted by the software visualizations to help users

*with the analysis of memory consumption?”* We generated a classification scheme according to the sources from which various data were collected. We found that all software visualizations display data from the program execution. The most popular data extracted from program execution involves memory events (allocations, accesses, releases) and references between variables. About 47.82% of software visualizations show static information. Only one visual approach displays information about changes between versions. Additionally, our findings show that most visualizations dismiss connecting the information from program execution with information from source code, like lines of code or classes. We consider that illustrating and piecing together information from both sources reduces the effort of practitioners to analyze memory consumption.

- *“Which software visualizations have been proposed to help users with the analysis of memory consumption?”* We categorized the articles according to the classification scheme proposed by Keim [66]. As a result, we discovered that 60.87% of software visualizations employ one visual technique. The most popular technique is the geometrically-transformed display, frequently used in articles that propose node-link diagrams to illustrate relationships between elements. We also detected that most software visualizations provide interaction mechanisms. However, the visualization mantra “Overview first, zoom and filter, then details on demand” [131] is not always considered by the current visualizations. Finally, most software visualizations are displayed using a standard monitor. Therefore, it will be valuable to implement visualizations that use other mediums, such as tactile devices or 3D environments, and study their effectiveness in the field.
- *“How are software visualizations to help users with the analysis of memory consumption evaluated?”* We classified the selected studies based on the work of Merino et al. [84]. Our results indicate that most software visualizations are evaluated empirically. About 56.52% of articles provide usage scenarios as an evaluation strategy and use popular benchmarks like *DaCapo suite* [21], *DB suite*, *Reptile* [153], *GCOld* [107], *Paraffins*, or open-source projects to demonstrate the benefits of the proposed visualization. Nonetheless, most articles lack robust empirical evaluation of how visualizations perform in practice with software developers and real-world applications.
- *“What software visualization tools or prototypes are available to help users with the analysis of memory consumption?”* We collected from the selected articles the tool’s name and the links from which the visualization tool is available. Only 21.73% of the articles provide an existing link. Additionally, we identified links to visualization tools for 23.91% of the articles due to web search. Overall, around 54.36% of the visualizations are not available. Consequently, we consider the lack of availability one of the main weak points in the field.

***Software visualization in practice.*** In Chapter 3, we introduced Vismep, an interactive visualization prototype to assist programmers in monitoring the memory usage of Python programs. Vismep summarizes how the program runs and allocates memory using polymetric views [74]. Vismep also connects the source code with dynamic information. Chapter 2 reports that these software aspects (the connection between source code and dynamic information, calling relationships between functions/methods) are not commonly displayed. Therefore, we

explored how valuable this information is for practitioners. We also presented an exploratory study to understand how Vismep supports eleven programmers in practice and the perceptions of programmers about the tool. This study answers the following research questions:

- “*How does Vismep support programmers when analyzing memory consumption?*” We found that participants 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 allocates 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. We also noticed that Participants used different Vismep views or combined some of them to obtain the required information. Furthermore, we noticed that participants considered some views more suitable based on the activities performed. For example, they often used the *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 to use Vismep to perform the activities. We detected missing information that users required and possibilities to improve the design of Vismep and other tools.
- “*How do programmers perceive Vismep when analyzing memory consumption?*” We reported that participants positively perceived Vismep because they indicated a low to moderate mental workload effort when using it and estimated that Vismep offers high overall usability. Furthermore, we also detected positive and negative aspects that participants mentioned about Vismep. These aspects open the door to potential improvements and points to consider for supporting programmers, as well as, the factors (*e.g.*, easy to use, intuitive, useful) that participants desire from a tool.

***Programmer needs and tool usage.*** In Chapter 4, we extended our interest to study participants’ questions when analyzing memory usage and how well Vismep and Tracemalloc address those needs. Therefore, we conducted an exploratory study with twenty-two programmers analyzing the memory consumption of Python applications with which they were familiar using Vismep and Tracemalloc. From our observations, we provided (i) an empirically-based set of questions that programmers ask during memory usage analysis and (ii) a report about how programmers answer those questions with Vismep and Tracemalloc. As far as we know, our study is the first to highlight some challenges and express practicability concerns when analyzing memory consumption. This study answers the following research questions:

- “*What questions do Python programmers ask when analyzing memory usage?*” We found 34 different questions raised by participants and organized them into five categories based on the information needed and the programmer’s behavior: (i) understanding source code, (ii) understanding control flow, (iii) discovering the memory usage at a single point of time, (iv) comparing and contrasting memory usage, and (v) discovering memory events. We included questions about understanding source code and control

flow because participants asked them to answer questions from the rest of the categories. Our results show that programmers require support for software comprehension when addressing a memory issue or determining the root cause of a potential failure. Since our study is the first centered on the needs programmers have when analyzing memory usage, we reported (i) asked questions that were not reported in previous studies on developer information needs, and (ii) a valuable information to guide researchers in designing tools that adequately support programmers in this context.

- “*How do Python programmers answer these questions using Vismep and Tracemalloc?*” We provided an observational analysis about how programmers employ *Vismep* and *Tracemalloc* to answer the raised questions. We discovered that participants numerous times need to combine multiple views (*e.g.*, Call graph view and Source code view) from *Vismep* or use multiple API calls from *Tracemalloc* to obtain the required information. We noticed that answering most questions of our study is generally well supported by *Vismep* and *Tracemalloc*. However, participants often had difficulties with these tools in some activities (i) searching entities, (ii) understanding control flow in a particular context, (iii) understanding memory usage at a fine-grained level (line of code, instance), (iv) connecting static and dynamic information, (v) comparing dynamic information at different levels (*e.g.*, between executions and versions) and (vi) exploring creation, release and the state of data structures. Consequently, we consider that programmers require more comprehensive support in (i) comparing and piecing information together and (ii) connecting dynamic information with source code.

## 5.2 Limitations

**Literature review.** As far as we know, the literature review presented in this thesis is the first one focused on software visualizations that help the user to comprehend memory usage. We conducted our study following guidelines for systematic reviews [69, 70] and considering the main characteristics of surveys [81, 83, 84, 85, 105, 106, 119] focused on software visualizations. However, our study and results are subject to certain limitations:

- *Search of articles.* Our study may not cover all the relevant articles in the field. Although we performed a systematic search based on various steps, our search query is biased by the specific keywords of this set of articles. We decided to decrease this threat by performing an additional manual search and bi-directional snowballing. These two phases assisted in our finding some missing relevant studies.
- *Selection of articles.* A relevant article may be excluded during the selection phase and vice versa. We specified inclusion/exclusion criteria and a quality assessment to reduce bias in selecting articles. We also calculated inter-rater reliability metrics among reviewers and held meetings to discuss discrepancies.
- *Data extraction.* The data extraction process could be biased. To reduce this threat, we defined a protocol to collect each article’s data. The thesis author managed a spreadsheet to keep records of relevant text segments and recognize irregularities like missing information. The two thesis supervisors checked if the data gathered was correct.

- *Data analysis.* The data analysis process could be biased. The thematic analysis process includes generating codes and defining themes (patterns) that allow answering the research questions. The codes and themes generated differ depending on the coder’s experience, point of view, and level of abstraction. For example, to respond to RQ1, we detected visualizations focused on analyzing specific points. However, some articles do not present clear or explicit objectives; therefore, we created a general-purpose group to add articles with no specific pattern. We tried to reduce this threat by checking the consistency of the process. Due to this, the two thesis supervisors examined the description of themes and the data coding. We held three discussion meetings to analyze the codes and the themes generated. As a result, we solve the differences.

***Software visualization in practice.*** We introduced Vismep, an interactive visualization prototype to assist programmers in analyzing the memory usage of Python programs. Vismep summarizes how the program runs and allocates memory using polymetric views [74]. Vismep also connects the source code with dynamic information. We also examined how valuable this information is for practitioners with an exploratory study involving eleven programmers. This study and the findings are subject to some limitations:

- *Information of Vismep.* Vismep highlights the memory usage per function and line-by-line to point out memory allocations. It also connects the source code with information from program execution, something that most visualizations dismiss [23], since we want to know the impact of this information on program comprehension. However, Vismep does not show information provided by other approaches [23], such as objects created, garbage collection information, and memory usage over time. Therefore, our results could be different considering this kind of information. We try to reduce this threat by collecting information widely reported by current tools Chapter 3.
- *Vismep visual design.* Although Vismep does not introduce a novel visualization technique, it adequately combines demonstrated techniques, such as polymetric views [74], the node-link diagram and the scatter plot. We decided to use a linked node because they are widely utilized to represent calling relationships between functions in software visualizations centered on assisting memory consumption analysis (see Chapter 2). Therefore, since we wanted to understand how well the current tools support some practitioners, we used a component similar to the ones in the current tools. Secondly, linked node diagrams are commonly used to represent relationships between elements, so it is a visual representation that is simple to understand and does not imply a very high learning curve [14]. However, the larger the graph, the more complex it is to understand the visualization. Consequently, we opted for creating the Sub call graph view to reduce complexity and allow users to focus on specific elements. Furthermore, in Chapter 2, we noticed alternatives to visualize call graphs, however, there is no empirical evidence of which visual representation is better for this purpose. Regarding the decision to use visual properties, the idea was to use polymetric views [74] to help users identify exceptional entities (*e.g.*, hotspot allocations, unexpected memory usage) based on the visual attributes [42]. Nonetheless, there is still a room for improvement in visualization (*e.g.*, filtering and navigating).

- *Selection of participants and applications.* 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.
- *Data extraction.* The data extraction could be biased. During the data collection and transcription process for RQ1, spreadsheets that summarize the work sessions were generated. Tools were employed to generate subtitle files from the recordings, which subsequently go through a process of revising and improving the poorly written parts. The YouTube Marks<sup>1</sup> tool was also used to facilitate the analysis of videos by placing tags and adding comments to those tags in order to divide and identify the parts of the sessions more easily. Additionally, the thesis author checked if the spreadsheets generated were consistent with the audio, video records, and tracking logs to minimize biases during the process.
- *Data analysis.* The data analysis process could be biased. For the data analysis of this study, the thesis author identified the information needs. However, identifying information needs based on a participant’s behavior can be inaccurate since they do not always verbalize their thoughts explicitly. To minimize the inaccuracy in the process, a thesis supervisor contrasted different scenarios and the use of Vismep to satisfy the information needs. This supervisor also checked if the information needs identified were consistent with the information in the spreadsheets.
- *Generalization of results.* The difficulty of conducting this type of large-scale study restricts the generalization of our results. For instance, transcribing and obtaining the information needed for a session requires considerable time. This study presents a number of sessions similar to some prior works (*e.g.*, Fernandez [22] with eight sessions), and the participants’ variability may represent this tool’s end users. The generalization of results is limited due to the number of participants, the lack of diversity in tasks, and the duration of the sessions. In addition, involving participants who voluntarily participate in the study and choose applications with specific memory issues according to the study goals takes time and effort.

***Programmer needs and tool usage.*** We research programmers’ questions when analyzing memory usage and how well Vismep and Tracemalloc address those needs. We conducted an exploratory study with twenty-two programmers analyzing the memory consumption of Python applications with which they were familiar using Vismep and Tracemalloc. This study and the findings are subject to some limitations:

- *Study design.* We asked the same open questions proposed in Chapter 3 since we aimed that participants had a goal they cared about and looked for the information they considered valuable to understand memory usage and detect optimization opportunities.

---

<sup>1</sup><https://github.com/tinchodias/youtube-marks>



Thus, we considered that providing defined tasks (*e.g.*, select allocation hotspots) instead of these questions would prevent participants from naturally defining a goal they care about when analyzing memory usage. Furthermore, we expected that dividing participants into two groups (G1 and G2) and analyzing the same application with Vismep and Tracemalloc would increase the variety and range of the practitioners' questions. We also considered that programmers might analyze memory usage and address memory issues using several tools since they require various features not necessarily supported by a single tool, as shown in Chapter 3. Note that our goal has not been to compare Vismep and Tracemalloc but to explore how programmers use these tools when analyzing memory usage. However, an empirical study comparing Vismep and Tracemalloc would be valuable for future research.

- *Data extraction.* The data extraction could be biased. The student supervised by the thesis author performed the data collection and transcription of each work session. Then, the thesis author checked the generated spreadsheets based on the video recordings and event logs to minimize inconsistencies. Furthermore, the thesis author conducted the steps to identify the questions asked by the participants. Identifying concrete questions based on user behavior may be inaccurate, as participants did not explicitly verbalize their thoughts. Finally, the process of defining general questions may suffer from uncertainty. For instance, it can be challenging to distinguish question 23 “*What will be the impact in memory consumption change?*” from question 26 “*What is the difference in memory consumption between these code executions?*”. To minimize inaccuracy in the inference process, the thesis author contrasted different scenarios and events related to the same questions and checked if the inferred questions were consistent with the information from spreadsheets. All the authors held two meetings to discuss any discrepancy in the inferred questions' consistency.
- *Data analysis.* Regarding the classification scheme, the thesis author conducted a thematic analysis to organize the inferred questions based on the information needed and the programmer's behavior in answering a question. The codes and themes generated vary depending on the coder's experience, level of abstraction, and point of view. To mitigate this threat, two supervisors of this thesis checked the consistency of the process by examining the description of themes with the associated data. We conducted two meetings to discuss and resolve the disagreements among generated codes and themes. Additionally, two reviewers independently categorized the inferred questions using the classification scheme, and a measure of agreement between reviewers was calculated to validate the reliability.
- *Generalization of results.* The selected memory profiler tools and the individual differences among participants influence the programmers' questions and how they answer those. Consequently, our data could differ given a distinct set of memory profiler tools or participants. The latter must be considered when interpreting or generalizing the results. We mitigated this threat by selecting programmers with different backgrounds and experience levels. In addition, we opted for memory profiler tools that provide diverse information and report presentations commonly offered by other tools for memory usage analysis. We also discussed and analyzed the tool support for answering questions considering the features of the tools. As a result, we noticed that many of the inferred questions were independent of whether they could be answered with the tools or not.

Furthermore, no new questions were detected in the last work sessions. The latter suggests that some of our results will likely generalize to other Python tools. A follow-up study in which participants were asked to work with another set of memory profiler tools would help demonstrate that one or more questions have more or no support than noted in our analysis.

### 5.3 Empirical Foundation Impact

**Literature review.** Software visualizations are designed considering several dimensions (*e.g.*, data abstracted, visual representation) to support developers when performing a task. Organizing and collecting software visualizations helps (i) practitioners to find a suitable visualization for their task, and (ii) researchers analyze the state-of-art and detect challenges or research areas that are worth exploring [81, 83, 84, 85, 105, 106, 119]. To the best of our knowledge, our work is the first literature review of software visualizations focused on supporting the user to comprehend memory consumption.

We conclude that our findings are valuable in guiding new research and facilitating practitioners to find available software visualizations. Our study also shows if aspects mentioned in prior work have been considered or not through time. For instance, prior work [83, 84] published between 2016 and 2018 mentioned that evaluation is weak in software visualizations and that this needs to be worked. However, our results indicate that most articles lack evidence regarding how visualizations perform in practice, including articles published after 2018. We consider that some points need to be considered to progress more in this topic (see Section 5.4).

Furthermore, we noticed that software visualization has not fully explored domain-specific memory analysis, memory issue identification, and memory regression analysis. Recent publications [77, 79, 110] are moving in this direction; therefore, our research shows research areas worth exploring in the right direction. Similarly, we detected that employing another medium (*e.g.*, 3D immerse environment, wall-display) than a single monitor screen could be attractive since prior work shows the relevance of medium in software visualizations [86]. The latter could be in the right direction due to the increment of recent publications using other mediums [59, 91, 92].

**Software visualization in practice.** One of the goals of this thesis is to explore the following points considered at Chapter 2: (i) the benefit of displaying aspects from software related to how the program runs and mapping the source code with dynamic information, (ii) how a visualization perform in practice, and (iii) the programmers' needs during memory usage analysis. To achieve this, we introduced Vismep, an interactive visualization prototype to assist practitioners in analyzing the memory usage of Python programs. Vismep was designed considering the aspects of the software mentioned above to study how valuable this information is for practitioners. In addition, we also conducted an exploratory study to understand how Vismep supports eleven programmers in practice and the perceptions of programmers about the tool.

This work provides empirical evidence about the benefits of visualizing aspects from dynamic and static information and connecting the source code with dynamic information. This evidence also points out how programmers use Vismep to analyze the memory usage of Python programs (familiar code) and obtain information that they consider valuable. Therefore, this study is a stepping stone for further researchers to report visualization usage and examine whether the visualization helps programmers obtain the required information or struggle in the process. Furthermore, the limitations of this study and the experience of carrying it out made us notice some points that need to be considered to improve empirical evaluations in the field (see Section 5.4).

***Programmer needs and tool usage.*** This dissertation also presents an exploratory study with twenty-two programmers monitoring the memory usage of Python programs with which they were familiar using Vismep and Tracemalloc. We analyzed the information programmers need about the software application when analyzing memory usage since it is key to better support this activity and observing how current tools satisfy these needs. As a result, we present (i) an empirically-based set of questions that programmers ask during memory usage analysis and (ii) a report about how programmers answer those questions with Vismep and Tracemalloc.

We conclude that our findings represent empirical support to confirm some assumptions of prior work [22, 154, 156]. We also consider that our reports about programmers' needs while monitoring memory usage would help improve the design of the current and new tools, as described in Chapter 4. Additionally, further researchers could use our results to (i) recognize how well a tool satisfies the programmers' needs, and (ii) propose a tool supporting needs that may not currently be covered. Finally, our results could facilitate the organization of current approaches to help practitioners find a suitable tool for their needs (see Section 5.4).

## 5.4 Future Work

This dissertation provides incentives for follow-up studies, which we formulate in this section.

***Organizing visualizations.*** As mentioned in Chapter 2, we could expand our classification of visualizations by considering the programmers' needs found in Chapter 4. Therefore, this future work could organize the data more accurately to identify which tasks support visualizations and detect which current approaches are suitable to satisfy certain programmers' needs. Another further research that may be helpful is to generate a software visualization ontology [87] based on the information extracted in Chapter 2 and by updating the review to add recent publications [24, 26] and other data (*e.g.*, requirements, if it is maintained) from tools. Based on this ontology, a tool could be implemented that is capable of visualizing, filtering, and updating information. Consequently, this work could support (i) practitioners in finding a suitable tool based on diverse aspects (*e.g.*, tasks supported, data illustrated, require execution environment) and (ii) researchers in locating visualizations that could be baselines for an experiment and exploring state-of-art.

***Evaluating visualizations.*** We found that some studies focused on visualizing memory

usage adopted an empirical strategy to evaluate their approaches (see Chapter 2). However, we noticed that most articles lack robust empirical evaluation that involves software developers. We also detected, based on our exploratory studies from Chapter 3 and Chapter 4, that conducting empirical studies with participants of the target audience could be complex since several aspects must be considered:

- *Tasks.* The software visualizations studied in this dissertation are proposed to support programmers in tasks dealing with concerns related to memory usage analysis. We found in Chapter 2 that little is known about the information programmers need when analyzing memory usage. Consequently, the goals of proposed visualizations may differ from the developers' current needs since we noticed in Chapter 2 that some visualizations are based on authors' personal experiences or assumptions without empirical backup. The studies described in Chapter 3 and Chapter 4 provide empirical evidence about the information programmer needs. As a result, our findings could be used to confirm and support design decisions and improve evaluations in software visualizations by considering the needs of programmers. However, as mentioned in the previous chapters, a wide range of further studies could be conducted to explore the needs of programmers in this field fully. For instance, analyze the programmers' needs under more specific situations (*e.g.*, to solve particular memory issues, analysis of memory usage in particular architectures) and in a more controlled environment.
- *Participants.* Some studies recommended conducting experiments with participants of the target audience [84, 163]. In our case, to improve the evaluations for software visualizations, the nature of the problem domain may require expert developers with a high level of knowledge regarding memory management [156]. For instance, to evaluate visualizations focused on analyzing cache performance or applications with certain architecture (*e.g.*, HPC), participants should understand the illustrated information to perform the respective tasks. Therefore, the studies must consider the goal of the evaluation to define how to select the participants (*e.g.*, novice programmers, experts in performance). We consider that studying the aspects of participants for studies in the field could be interesting [132, 164].
- *Applications under study.* We noticed in Chapter 2 that articles present reports of using their proposed visualization to analyze various software programs or projects. However, several studies do not explicitly describe the context or situation in which these projects were analyzed, and many are not openly available as open-source projects. Therefore, it is challenging to replicate evaluations and have a standardized way of evaluating visualizations to promote their comparison. Consequently, further work that would be valuable is creating a project set (*i.e.*, projects with particular memory issues) to facilitate researchers gathering specific data from particular projects and promote evaluations that ease comparison across tools.
- *Aspects to be evaluated.* As mentioned before, it is relevant to explicitly define the evaluation's goal and define which aspects of the visualizations are evaluated. In Chapter 3 and Chapter 4, we center on usefulness, cognitive load, and usability. In future work, we plan to study other aspects (*e.g.*, time, correctness, recollection, emotions) to expand the assessment of software visualizations.

**Visualizing memory usage.** In Chapter 3, we introduced Vismep and explored how valuable Vismep is for practitioners when analyzing the memory usage of Python applications by conducting an explorative study. Our results show (i) the benefits of visualizing certain aspects of software, (ii) how programmers use Vismep to find relevant information for assessing memory usage, and (iii) the perceptions of programmers about Vismep. Furthermore, we also detected missing information that users required and suggestions about how to improve the design of Vismep. Consequently, it would be beneficial to (i) collect and visualize some information (*e.g.*, memory releases, memory accesses) that Vismep does not show currently and (ii) explore other kinds of visual techniques and interactions to enhance the visualization of Vismep. Additionally, conducting studies focused on developing specialized visual tools to help users face and repair particular memory issues (*e.g.*, leaks, churn) would contribute to expanding the field’s opportunities.

**Missed opportunities and other aspects.** In Chapter 4, we reported the questions that programmers ask themselves when analyzing the memory usage of familiar software projects. We also described how they used Vismep and Tracemalloc to answer those questions. In addition, we summarize when participants had difficulties during the process. However, it would be valuable to document the situations systematically and propose a guide system [155] as tool improvement to help users to take advantage of the information that tools provide.

Furthermore, conducting studies where the variables are more controlled (controlled experiments) would be valuable because it would extend our research and provide more information about the behavior of programmers when analyzing memory consumption. For instance, we could explore whether certain factors related to the developers’ profiles concerning background and experience may influence tool usage or the asked questions. Another example for future work is generating a ranking of the questions asked for a particular situation, as well as, examining situations related to addressing memory issues.

**Analyzing other languages.** Finally, our results are still limited due to the selected programming language. We plan to conduct more studies considering other languages (*e.g.*, Java, Javascript) to explore the differences and similarities of tools usage, questions asked, and perceptions.

# Bibliography

- [1] Guppy 3 home page. Accessed: 2022-01-27.
- [2] Identifying memory leaks - pympler documentation. Accessed: 2022-01-27.
- [3] Memory-profiler. Accessed: 2022-01-27.
- [4] memray, a memory profiler for python. Accessed: 2022-01-27.
- [5] A python memory profiler for data processing and scientific computing applications. Accessed: 2022-01-27.
- [6] Python object graphs. Accessed: 2022-01-27.
- [7] Tracemalloc - trace memory allocations. Accessed: 2022-01-27.
- [8] vprof: Visual profiler for python. Accessed: 2022-01-27.
- [9] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS '10*, page 53–62, New York, NY, USA, 2010. Association for Computing Machinery.
- [10] Abdullah Al-Sakkaf, Mazni Omar, and Mazida Ahmad. A systematic literature review of student engagement in software visualization: a theoretical perspective. *Computer Science Education*, 29(2-3):283–309, 2019.
- [11] Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–9. IEEE, 2013.
- [12] Aaron Bangor, Philip T. Kortum, and James T. Miller. An empirical evaluation of the system usability scale. *International Journal of Human–Computer Interaction*, 24(6):574–594, Jul 2008.
- [13] Sarita Bassil and Rudolf K Keller. Software visualization tools: Survey and analysis. In *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*, pages 7–17. IEEE, 2001.

- [14] Fabian Beck, Michael Burch, Stephan Diehl, and Daniel Weiskopf. A taxonomy and survey of dynamic graph visualization. In *Computer graphics forum*, volume 36, pages 133–159. Wiley Online Library, 2017.
- [15] Laura Beckwith, Margaret Burnett, Susan Wiedenbeck, Curtis Cook, Shraddha Sorte, and Michelle Hastings. Effectiveness of end-user debugging software features: Are there gender issues? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, page 869–878, New York, NY, USA, 2005. Association for Computing Machinery.
- [16] Laura Beckwith, Cory Kissinger, Margaret Burnett, Susan Wiedenbeck, Joseph Lawrance, Alan Blackwell, and Curtis Cook. Tinkering and gender in end-user programmers' debugging. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, page 231–240, New York, NY, USA, 2006. Association for Computing Machinery.
- [17] Laure Bedu, Olivier Tinh, and Fabio Petrillo. A tertiary systematic literature review on software visualization. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 33–44. IEEE, 2019.
- [18] David Beniamine, Matthias Diener, Guillaume Huard, and Philippe O. A. Navaux. Tabarnac: Visualizing and resolving memory access issues on numa architectures. In *Proceedings of the 2nd Workshop on Visual Performance Analysis, VPA '15*, New York, NY, USA, 2015. Association for Computing Machinery.
- [19] Alexandre Bergel, Abhinav Bhatele, David Boehme, Patrick Gralka, Kevin Griffin, Marc-André Hermanns, Dušan Okanović, Olga Pearce, and Tom Vierjahn. Visual analytics challenges in analyzing calling context trees. In *Programming and Performance Visualization Tools: International Workshops, ESPT 2017 and VPA 2017, Denver, CO, USA, November 12 and 17, 2017, and ESPT 2018 and VPA 2018, Dallas, TX, USA, November 16 and 11, 2018, Revised Selected Papers 6*, pages 233–249. Springer, 2019.
- [20] Emery D. Berger. Scalene: Scripting-language aware profiling for python, 2020.
- [21] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41(10):169–190, October 2006.
- [22] Alison Fernandez Blanco, Juan Pablo Sandoval Alcocer, and Alexandre Bergel. Effective visualization of object allocation sites. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 43–53. IEEE, 2018.
- [23] Alison Fernandez Blanco, Alexandre Bergel, and Juan Pablo Sandoval Alcocer. Software visualizations to analyze memory consumption: A literature review. *ACM Comput. Surv.*, 55(1), jan 2022.

- [24] Alison Fernandez Blanco, Alexandre Bergel, Juan Pablo Sandoval Alcocer, and Araceli Queirolo Córdoba. Visualizing memory consumption with vismep. In *2022 Working Conference on Software Visualization (VISSOFT)*, pages 108–118, 2022.
- [25] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. *SIGPLAN Not.*, 43(10):109–126, October 2008.
- [26] Jan H. Boockmann and Gerald Lüttgen. Shape-analysis driven memory graph visualization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC '22*, page 298–308, New York, NY, USA, 2022. Association for Computing Machinery.
- [27] Stuart Byma and James R. Larus. Detailed heap profiling. *SIGPLAN Not.*, 53(5):1–13, jun 2018.
- [28] Andrew M Cheadle, AJ Field, JW Ayres, Neil Dunn, Richard A Hayden, and J Nystrom-Persson. Visualising dynamic memory allocators. In *Proceedings of the 5th International Symposium on Memory Management, ISMM '06*, page 115–125, New York, NY, USA, 2006. Association for Computing Machinery.
- [29] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O’Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *Proceedings of the 25th European Conference on Object-oriented Programming, ECOOP'11*, pages 383–407, Berlin, Heidelberg, 2011. Springer-Verlag.
- [30] Noptanit Chotisarn, Leonel Merino, Xu Zheng, Supaporn Lonapalawong, Tianye Zhang, Mingliang Xu, and Wei Chen. A systematic literature review of modern software visualization. *Journal of Visualization*, 23(4):539–558, 2020.
- [31] ANM Imroz Choudhury, Kristin C Potter, and Steven G Parker. Interactive visualization for memory reference traces. In *Computer Graphics Forum*, volume 27, pages 815–822. Wiley Online Library, 2008.
- [32] ANM Imroz Choudhury and Paul Rosen. Abstract visualization of runtime memory behavior. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8. IEEE, 2011.
- [33] ANM Imroz Choudhury, Bei Wang, Paul Rosen, and Valerio Pascucci. Topological analysis and visualization of cyclical behavior in memory reference traces. In *2012 IEEE Pacific Visualization Symposium*, pages 9–16. IEEE, 2012.
- [34] James Clause and Alessandro Orso. Leakpoint: Pinpointing the causes of memory leaks. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, page 515–524, New York, NY, USA, 2010. Association for Computing Machinery.
- [35] Juliet M Corbin and Anselm C Strauss. *Basics of qualitative research*. SAGE Publications, Thousand Oaks, CA, 3 edition, January 2008.
- [36] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.



- [37] Andrew R Dalton and William Krehling. Automated construction of memory diagrams for program comprehension. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, New York, NY, USA, 2010. Association for Computing Machinery.
- [38] Brian De Alwis and Gail C Murphy. Answering conceptual queries with ferret. In *Proceedings of the 30th international conference on Software engineering*, pages 21–30, 2008.
- [39] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Software Visualization*, pages 151–162. Springer, 2002.
- [40] Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in java. In Rachid Guerraoui, editor, *ECOOP' 99 — Object-Oriented Programming*, pages 116–134, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
- [41] Joanna F DeFranco and Phillip A Laplante. A content analysis process for qualitative software engineering research. *Innovations in Systems and Software Engineering*, 13(2):129–141, 2017.
- [42] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-oriented reengineering patterns*. Elsevier, 2002.
- [43] Stephan Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer Science & Business Media, 2007.
- [44] Andi Drebes, Antoniu Pop, Karine Heydemann, and Albert Cohen. Interactive visualization of cross-layer performance anomalies in dynamic task-parallel applications and systems. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 274–283. IEEE, 2016.
- [45] S. Ducasse, M. Lanza, and R. Bertuli. High-level polymetric views of condensed run-time information. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 309–318, 2004.
- [46] Fleur Duseau, Bruno Dufour, and Houari Sahraoui. Vasco: A visual approach to explore object churn in framework-intensive applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 15–24. IEEE, 2012.
- [47] Satu Elo and Helvi Kyngäs. The qualitative content analysis process. *Journal of advanced nursing*, 62(1):107–115, 2008.
- [48] Alison Fernandez Blanco, Alexandre Bergel, Juan Pablo Sandoval Alcocer, and Araceli Queirolo Cordova. Visualizing memory consumption with vismep, August 2022.
- [49] Joseph L Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.
- [50] Joseph L Fleiss, Bruce Levin, and Myunghee Cho Paik. *Statistical methods for rates and proportions*. john wiley & sons, 2013.

- [51] Thomas Fritz and Gail C Murphy. Using information fragments to answer the questions developers ask. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 1, pages 175–184. ACM Press, 2010.
- [52] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. Memory and resource leak defects and their repairs in java projects. *Empirical Software Engineering*, 25(1):678–718, 2020.
- [53] Alfredo Giménez, Todd Gamblin, Ilir Jusufi, Abhinav Bhatele, Martin Schulz, Peer-Timo Bremer, and Bernd Hamann. Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. *IEEE transactions on visualization and computer graphics*, 24(7):2180–2193, 2018.
- [54] Patrick Gralka, Christoph Schulz, Guido Reina, Daniel Weiskopf, and Thomas Ertl. Visual exploration of memory traces and call stacks. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 54–63, 2017.
- [55] Rebecca 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):1727–1731, 2015.
- [56] Sandra G Hart and Lowell E Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. In *Advances in psychology*, volume 52, pages 139–183. Elsevier, 1988.
- [57] Morten Hertzum. Reference values and subscale patterns for the task load index (tlx): a meta-analytic review. *Ergonomics*, 64(7):869–878, 2021.
- [58] Jeisson Hidalgo-Céspedes, Gabriela Marín-Raventós, and Vladimir Lara-Villagrán. Learning principles in program visualizations: a systematic literature review. In *2016 IEEE frontiers in education conference (FIE)*, pages 1–9. IEEE, 2016.
- [59] Adrian Hoff, Lea Gerling, and Christoph Seidl. Utilizing software architecture recovery to explore large-scale software systems in virtual reality. In *2022 Working Conference on Software Visualization (VISSOFT)*. IEEE, 2022.
- [60] Andreas Holzinger. Usability engineering methods for software developers. *Commun. ACM*, 48(1):71–74, jan 2005.
- [61] Huihui Nora Huang, Eric Verbeek, Daniel German, Margaret-Anne Storey, and Martin Salois. Atlantis: Improving the analysis and visualization of large assembly execution traces. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 623–627. IEEE, 2017.
- [62] Alejandro Infante and Alexandre Bergel. Efficiently identifying object production sites. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 575–579. IEEE, 2015.
- [63] Ryosuke Ishizue, Kazunori Sakamoto, Hironori Washizaki, and Yoshiaki Fukazawa. Pvc.js: visualizing c programs on web browsers for novices. *Heliyon*, 6(4):e03806, 2020.

- [64] Essi Isohanni and Hannu-Matti Järvinen. Are visualization tools used in programming education? by whom, how, why, and why not? In *Proceedings of the 14th Koli Calling International Conference on Computing Education Research*, Koli Calling '14, page 35–40, New York, NY, USA, 2014. Association for Computing Machinery.
- [65] Kamil Jezek and Richard Lipka. Antipatterns causing memory bloat: A case study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 306–315. IEEE, 2017.
- [66] Daniel A Keim. Information visualization and visual data mining. *IEEE transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002.
- [67] Taimur Khan, Henning Barthel, Achim Ebert, and Peter Liggesmeyer. Visualization and Evolution of Software Architectures. In *Visualization of Large and Unstructured Data Sets: Applications in Geospatial Planning, Modeling and Engineering - Proceedings of IRTG 1131 Workshop 2011*, volume 27, pages 25–42, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [68] Holger M Kienle and Hausi A Muller. Requirements of software visualization tools: A literature survey. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 2–9. IEEE, 2007.
- [69] Barbara Kitchenham. Procedures for performing systematic reviews. *Keele, UK, Keele University*, 33(2004):1–26, 2004.
- [70] Barbara Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.
- [71] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *29th International Conference on Software Engineering (ICSE'07)*, pages 344–353. IEEE, May 2007.
- [72] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. The road to live programming: insights from the practice. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 1090–1101. IEEE, 2018.
- [73] Juraj Kubelka, Romain Robbes, and Alexandre Bergel. Live programming and software evolution: Questions during a programming change task. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 30–41. IEEE, May 2019.
- [74] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.
- [75] Thomas D LaToza and Brad A Myers. Developers ask reachability questions. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 185–194. ACM Press, 2010.

- [76] Thomas D LaToza and Brad A Myers. Hard-to-answer questions about code. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10. Association for Computing Machinery, New York, NY, USA, 2010.
- [77] Haiyi Liu, Shaoying Liu, Chenglong Wen, and W Eric Wong. Tbem: Testing-based gpu-memory consumption estimation for deep learning. *IEEE Access*, 10:39674–39680, 2022.
- [78] David Lo, Nachiappan Nagappan, and Thomas 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, page 415–425, New York, NY, USA, 2015. Association for Computing Machinery.
- [79] Chang Lou, Cong Chen, Peng Huang, Yingnong Dang, Si Qin, Xinsheng Yang, Xukun Li, Qingwei Lin, and Murali Chintalapati. {RESIN}: A holistic service for dealing with memory leaks in production cloud infrastructure. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 109–125, 2022.
- [80] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40(6):190–200, jun 2005.
- [81] Jonathan I Maletic, Andrian Marcus, and Michael L Collard. A task oriented view of software visualization. In *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*, pages 32–40. IEEE, 2002.
- [82] Mark Marron, Cesar Sanchez, Zhendong Su, and Manuel Fahndrich. Abstracting runtime heaps for program understanding. *IEEE Transactions on Software Engineering*, 39(6):774–786, 2012.
- [83] Anna-Liisa Mattila, Petri Ihanola, Terhi Kilamo, Antti Luoto, Mikko Nurminen, and Heli Väättäjä. Software visualization today: Systematic literature review. In *Proceedings of the 20th International Academic Mindtrek Conference*, pages 262–271, New York, NY, USA, 2016. Association for Computing Machinery.
- [84] Leonel Merino, Mohammad Ghafari, Craig Anslow, and Oscar Nierstrasz. A systematic literature review of software visualization evaluation. *Journal of systems and software*, 144:165–180, 2018.
- [85] Leonel Merino, Mohammad Ghafari, and Oscar Nierstrasz. Towards actionable visualisation in software development. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 61–70. IEEE, 2016.
- [86] Leonel Merino, Mario Hess, Alexandre Bergel, Oscar Nierstrasz, and Daniel Weiskopf. Perfvis: Pervasive visualization in immersive augmented reality for performance awareness. In *Companion of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, page 13–16, New York, NY, USA, 2019. Association for Computing Machinery.

- [87] Leonel Merino, Ekaterina Kozlova, Oscar Nierstrasz, and Daniel Weiskopf. Vison: An ontology-based approach for software visualization tool discoverability. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 45–55. IEEE Computer Society, 2019.
- [88] K. J. Millman and M. Aivazis. Python for scientists and engineers. *Computing in Science Engineering*, 13(2):9–12, 2011.
- [89] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming*, pages 77–97, Berlin, Heidelberg, 2009. Springer-Verlag.
- [90] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. *SIGPLAN Not.*, 42(10):245–260, oct 2007.
- [91] David Moreno-Lumbreras, Jesus M Gonzalez-Barahona, and Andrea Villaverde. Babiaxr: Virtual reality software data visualizations for the web. In *2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 71–74. IEEE, 2022.
- [92] David Moreno-Lumbreras, Roberto Minelli, Andrea Villaverde, Jesus M. Gonzalez-Barahona, and Michele Lanza. Codecity: A comparison of on-screen and virtual reality. *Information and Software Technology*, 153:107064, 2023.
- [93] Sergio Moreta and Alexandru Telea. Visualizing dynamic memory allocations. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 31–38. IEEE, 2007.
- [94] Tao Mu, Jie Tao, Martin Schulz, and Sally A McKee. Interactive locality optimization on numa architectures. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis ’03, page 133–ff, New York, NY, USA, 2003. Association for Computing Machinery.
- [95] Richard Müller and Dirk Zeckzer. Past, present, and future of 3d software visualization. pages 63–74, 2015.
- [96] Tamara Munzner. *Visualization analysis and design*. CRC press, 2014.
- [97] Colin Myers and David Duke. A map of the heap: Revealing design abstractions in runtime structures. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS ’10, page 63–72, New York, NY, USA, 2010. Association for Computing Machinery.
- [98] Lucas Leandro Nesi, Samuel Thibault, Luka Stanisic, and Lucas Mello Schnorr. Visual performance analysis of memory behavior in a task-based runtime on hybrid platforms. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 142–151. IEEE, 2019.
- [99] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’07, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.

- [100] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 562–571, 2013.
- [101] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. Software evolution visualization: A systematic mapping study. *Information and Software Technology*, 55(11):1860–1883, 2013.
- [102] Kristian Nybom, Adnan Ashraf, and Ivan Porres. A systematic mapping study on api documentation generation approaches. In *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 462–469. IEEE, 2018.
- [103] Yunrim Park and Carlos Jensen. Beyond pretty pictures: Examining the benefits of code visualization for open source newcomers. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 3–10, 2009.
- [104] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, 2015.
- [105] Blaine Price, Ronald Baecker, and Ian Small. An introduction to software visualization. *Software visualization*, pages 3–27, 1998.
- [106] Blaine A Price, Ronald M Baecker, and Ian S Small. A principled taxonomy of software visualization. *Journal of Visual Languages & Computing*, 4(3):211–266, 1993.
- [107] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the 2nd international symposium on Memory management*, pages 143–154, New York, NY, USA, 2000. Association for Computing Machinery.
- [108] Tony Printezis and Alex Garthwaite. Visualising the train garbage collector. *SIGPLAN Not.*, 38(2 supplement):50–63, June 2002.
- [109] Tony Printezis and Richard Jones. Gcsplay: An adaptable heap visualisation framework. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '02*, page 343–358, New York, NY, USA, 2002. Association for Computing Machinery.
- [110] Ju Qian, Xu Zhou, and Hui Zhou. Prioritising test scripts for the testing of memory bloat in web applications. *IET Software*, 16(3):317–330, 2022.
- [111] Boris Quaing, Jie Tao, and Wolfgang Karl. Yaco: A user conducted visualization tool for supporting cache optimization. In *International Conference on High Performance Computing and Communications*, pages 694–703. Springer, 2005.
- [112] Araceli Nicole Queirolo Córdova. Mejorar la usabilidad y efectividad de una herramienta de perfilamiento de memoria. 2021.
- [113] Derek Rayside and Lucy Mendel. Object ownership profiling: A technique for finding and fixing memory leaks. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE '07*, page 194–203, New York, NY, USA, 2007. Association for Computing Machinery.

- [114] Steven P Reiss. Visualizing java in action. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–ff, New York, NY, USA, 2003. ACM, Association for Computing Machinery.
- [115] Steven P Reiss. Visualizing the java heap to detect memory problems. In *Visualizing Software for Understanding and Analysis, 2009. VISSOFT 2009. 5th IEEE International Workshop on*, pages 73–80. IEEE, 2009.
- [116] Steven P Reiss and Manos Renieris. Jove: Java as it happens. In *Proceedings of the 2005 ACM symposium on Software visualization*, pages 115–124, New York, NY, USA, 2005. ACM, Association for Computing Machinery.
- [117] Nathan P. Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. Elephant tracks: Generating program traces with object death records. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, PPPJ '11*, page 139–142, New York, NY, USA, 2011. Association for Computing Machinery.
- [118] George G. Robertson, Trishul Chilimbi, and Bongshin Lee. Allocray: Memory allocation visualization for unmanaged languages. In *Proceedings of the 5th International Symposium on Software Visualization, SOFTVIS '10*, page 43–52, New York, NY, USA, 2010. Association for Computing Machinery.
- [119] G-C Roman and Kenneth C Cox. A taxonomy of program visualization systems. *Computer*, 26(12):11–24, 1993.
- [120] Paul Rosen. A visual approach to investigating shared and global memory behavior of cuda kernels. In *Computer Graphics Forum*, volume 32, pages 161–170. Wiley Online Library, 2013.
- [121] Hani Bani Salameh, Ayat Ahmad, and Ashraf Aljammal. Software evolution visualization techniques and methods-a systematic review. In *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, pages 1–6. IEEE, 2016.
- [122] J. Saldana. *The Coding Manual for Qualitative Researchers*. Core textbook. SAGE Publications, 2021.
- [123] Juan Pablo Sandoval Alcocer, Harold Camacho Jaimes, Diego Costa, Alexandre Bergel, and Fabian Beck. Enhancing commit graphs with visual runtime clues. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 28–32, 2019.
- [124] Sekhar R Sarukkai and Andrew Beers. Monitoring data-structure evolution in distributed message-passing programs. In *Proceedings of HICSS-29: 29th Hawaii International Conference on System Sciences*, volume 1, pages 310–319. IEEE, 1996.
- [125] Jeff Sauro. *A practical guide to the system usability scale: Background, benchmarks & best practices*. Measuring Usability LLC, 2011.
- [126] C.B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

- [127] Mariam Sensalire, Patrick Ogao, and Alexandru Telea. Evaluation of software visualization tools: Lessons learned. In *2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 19–26. IEEE, 2009.
- [128] Abderrahmane Seriai, Omar Benomar, Benjamin Cerat, and Houari Sahraoui. Validation of software visualization tools: A systematic mapping study. In *2014 Second IEEE Working Conference on Software Visualization*, pages 60–69. IEEE, 2014.
- [129] Mojtaba Shahin, Peng Liang, and Muhammad Ali Babar. A systematic review of software architecture visualization techniques. *Journal of Systems and Software*, 94:161–185, 2014.
- [130] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, 11(1):92–99, 1992.
- [131] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *The craft of information visualization*, Interactive Technologies, pages 364–371. Elsevier, 2003.
- [132] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334, 2014.
- [133] Jonathan Sillito, Gail C Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 23–34, New York, NY, USA, 2006. Association for Computing Machinery.
- [134] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, Jul 2008.
- [135] Rebecca Smith and Scott Rixner. Leveraging managed runtime systems to build, analyze, and optimize memory graphs. *ACM SIGPLAN Notices*, 51(7):131–143, 2016.
- [136] KR Srinath. Python—the fastest growing programming language. *International Research Journal of Engineering and Technology (IRJET)*, 4(12):354–357, 2017.
- [137] Amitabh Srivastava and Alan Eustace. *ATOM: A system for building customized program analysis tools*, volume 29. ACM, New York, NY, USA, 1994.
- [138] Jaishankar Sundararaman and Godmar Back. Hdpv: interactive, faithful, in-vivo runtime state visualization for c/c++ and java. In *Proceedings of the 4th ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2008. Association for Computing Machinery.
- [139] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.



- [140] Jie Tao, Wolfgang Karl, and Martin Schulz. Visualizing the memory access behavior of shared memory applications on numa architectures. In *International Conference on Computational Science*, pages 861–870. Springer, 2001.
- [141] Gareth Terry, Nikki Hayfield, Victoria Clarke, and Virginia Braun. Thematic analysis. *The Sage handbook of qualitative research in psychology*, pages 17–37, 2017.
- [142] Alfredo R Teyseyre and Marcelo R Campo. An overview of 3d software visualization. *IEEE transactions on visualization and computer graphics*, 15(1):87–105, 2009.
- [143] Scott Tilley and Shihong Huang. Documenting software systems with views iii: towards a task-oriented classification of program visualization techniques. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 226–233, New York, NY, USA, 2002. Association for Computing Machinery.
- [144] M. Tory and T. Moller. Human factors in visualization research. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):72–84, 2004.
- [145] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. Numamma: Numa memory analyzer. In *Proceedings of the 47th International Conference on Parallel Processing*, pages 1–10, New York, NY, USA, 2018. Association for Computing Machinery.
- [146] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, USA, 1986.
- [147] Muhammad Usman, Emilia Mendes, Francila Weidt, and Ricardo Britto. Effort estimation in agile software development: a systematic literature review. In *Proceedings of the 10th international conference on predictive models in software engineering*, pages 82–91, New York, NY, USA, 2014. ACM, Association for Computing Machinery.
- [148] Eric van der Deijl, Gerco Kanbier, Olivier Temam, and Elana D. Granston. A cache visualization tool. *Computer*, 30(7):71–78, 1997.
- [149] Renan Vasconcelos, Marcelo Schots, and Cláudia Werner. An information visualization feature model for supporting the selection of software visualizations. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 122–125, New York, NY, USA, 2014. Association for Computing Machinery.
- [150] Miguel Velez, Pooyan Jamshidi, Norbert Siegmund, Sven Apel, and Christian Kästner. On debugging the performance of configurable software systems: Developer needs and tailored tool support. *arXiv preprint arXiv:2203.10356*, 2022.
- [151] Raoul L Veroy, Nathan P Ricci, and Samuel Z Guyer. Visualizing the allocation and death of objects. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013.
- [152] Anthony J Viera, Joanne M Garrett, et al. Understanding interobserver agreement: the kappa statistic. *Fam med*, 37(5):360–363, 2005.
- [153] David Wakeling. Compiling lazy functional programs for the java virtual machine. *Journal of Functional Programming*, 9(6):579–603, 1999.

- [154] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. Analyzing data structure growth over time to facilitate memory leak detection. In *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, ICPE '19*, page 273–284, New York, NY, USA, 2019. Association for Computing Machinery.
- [155] Markus Weninger, Elias Gander, and Hanspeter Mössenböck. Guided exploration: A method for guiding novice users in interactive memory monitoring tools. *Proceedings of the ACM on Human-Computer Interaction*, 5(EICS):1–34, 2021.
- [156] Markus Weninger, Paul Grünbacher, Elias Gander, and Andreas 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.
- [157] Markus Weninger, Lukas Makor, and Hanspeter Mössenböck. Memory cities: Visualizing heap memory evolution using the software city metaphor. In *2020 Working Conference on Software Visualization (VISSOFT)*, pages 110–121, 2020.
- [158] Joao Werther, Glauco de Figueiredo Carneiro, and Rita Suzana Pitangueira Maciel. A systematic mapping on visual solutions to support the comprehension of software architecture evolution.
- [159] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 551–560, May 2011.
- [160] Benjamin Weyers, Christian Terboven, Dirk Schmidl, Joachim Herber, Torsten W Kuhlen, Matthias S Müller, and Bernd Hentschel. Visualization of memory access behavior on hierarchical numa architectures. In *2014 First Workshop on Visual Performance Analysis*, pages 42–49. IEEE, 2014.
- [161] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, pages 1–10, New York, NY, USA, 2014. Association for Computing Machinery.
- [162] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [163] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.
- [164] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.
- [165] Guoqing Xu and Atanas Rountev. Detecting inefficiently-used containers to avoid bloat. *SIGPLAN Not.*, 45(6):160–173, jun 2010.

- [166] Guoqing Xu and Atanas Rountev. Precise memory leak detection for java software using container profiling. *ACM Trans. Softw. Eng. Methodol.*, 22(3), jul 2013.
- [167] Ji Soo Yi, Youn ah Kang, John Stasko, and Julie A Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE transactions on visualization and computer graphics*, 13(6):1224–1231, 2007.
- [168] Yijun Yu, Kristof Beyls, and Erik H D’Hollander. Visualizing the impact of the cache on program execution. In *Proceedings Fifth International Conference on Information Visualisation*, pages 336–341. IEEE, 2001.
- [169] Marvin V Zelkowitz and Dolores R. Wallace. Experimental models for validating technology. *Computer*, 31(5):23–31, 1998.
- [170] Andreas Zeller and Dorothea Lütkehaus. Ddd—a free graphical front-end for unix debuggers. *ACM Sigplan Notices*, 31(1):22–27, 1996.
- [171] Thomas Zimmermann and Andreas Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.
- [172] Benjamin Zorn, Paul Hilfinger, et al. A memory allocation profiler for c and lisp programs. In *Proceedings of the Summer USENIX Conference*, pages 223–237. USENIX Association Berkeley, CA, USA, 1988.

# Annexes

# Annex A

## Search String for Digital Libraries

Table A.1 shows the search queries used for the three digital libraries.

Table A.1: Search query for the three digital libraries

Digital library	Search query
ACM	<i>Abstract:</i> ((software OR program OR application) AND (visualize OR visualization OR visualisation OR visualizations OR visualisations OR visuals OR visual) AND ("memory heap" OR "memory allocation" OR "memory consume" OR "memory consumption" OR "memory usage" OR "memory management" OR "memory issues" OR "memory issue" "memory bloats" OR "memory leaks" OR "memory access" OR "memory address"))
IEEE Xplore	("Abstract": "software" OR "Abstract": "program" OR "Abstract": "application") AND ("Abstract": "visualize" OR "Abstract": "visualization" OR "Abstract": "visualisation" OR "Abstract": "visualizations" OR "Abstract": "visualisations" OR "Abstract": "visuals" OR "Abstract": "visual") AND ("Abstract": "memory heap" OR "Abstract": "memory allocation" OR "Abstract": "memory consume" OR "Abstract": "memory consumption" OR "Abstract": "memory usage" OR "Abstract": "memory management" OR "Abstract": "memory issues" OR "Abstract": "memory bloats" OR "Abstract": "memory leaks" OR "Abstract": "memory access" OR "Abstract": "memory address")
Scopus	ABS ( ( software OR program OR application ) AND ( visualize OR visualization OR visualisation OR visualizations OR visualisations OR visuals OR visual ) AND ( "memory heap" OR "memory allocation" OR "memory consume" OR "memory consumption" OR "memory usage" OR "memory management" OR "memory issues" OR "memory issue" OR "memory bloats" OR "memory leaks" OR "memory access" OR "memory address" ) ) AND ( LIMIT-TO ( SUBJAREA , "COMP" ) ) AND ( LIMIT-TO ( DOCTYPE , "cp" ) OR LIMIT-TO ( DOCTYPE , "ar" ) ) AND ( LIMIT-TO ( LANGUAGE , "English" ) )

# Annex B

## Perception of Vismep and Tracemalloc

We provide information about the subjective mental workload and the usability perception experienced by participants using Vismep and Tracemalloc. Note that this information was extracted from the empirical study in Chapter 4, which aimed to identify questions that programmers ask when analyzing memory and how they use tools to answer them. Therefore, the data presented below cannot be used to compare Vismep and Tracemalloc since the study design does not consider that objective. However, it provides a stepping stone for future work.

### B.1 Cognitive Load

Table B.1 shows mean and standard deviation values of the overall and dimensions TLX scores for each group (G1 and G2) for Vismep and Tracemalloc. NASA-TLX score ranges from 0 (low mental workload) to 100 (high mental workload). Participants' average task load index using Vismep is 29.24 (std. dev. 14.34) for G1 and 30.76 (std. dev. 8.92) for G2. Therefore, according to Grier [55], and Hertzum [57], these measures indicate a low to moderate effort. The average cognitive load registered by participants using Tracemalloc is 36.67 (std. dev. 19.78) for G1 and 42.88 (std. dev. 24.24) for G2. As a consequence, these measures show moderate effort.

**Dimensions.** 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). Regardless of their group, participants perceived that mental demand, temporal demand, and effort means are the highest among all dimensions for Vismep. These scores are consistent with the challenges and problems reported in Section 4.5.6. In the case of Tracemalloc, participants, regardless of the group, registered that mental demand and effort means are the highest among all dimensions for Tracemalloc. These measures could reflect the challenging mental operations mentioned during the sessions since most participants struggle to navigate between dynamic information and code.

Table B.1: Means of overall workload and dimensions TLX scores for G1 and G2 using Vismep and Tracemalloc.

	G1				G2			
	Vismep		Tracemalloc		Vismep		Tracemalloc	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
<b>Overall TLX</b>	29.24	14.34	36.67	19.78	30.76	8.92	42.88	24.24
<b>Dimensions</b>								
Mental demand	43.63	25.79	46.36	31.71	42.73	17.94	57.27	29.36
Physical demand	17.27	20.54	21.82	27.50	17.27	20.04	32.73	29.36
Temporal demand	40	17.89	30	25.29	42.73	26.49	39.09	24.27
Performance	15.45	17.53	44.55	30.78	18.18	15.37	33.64	28.38
Effort	44.55	24.23	40.91	31.77	36.36	16.89	51.82	27.86
Frustration	14.55	16.95	36.36	27.67	27.27	24.94	42.73	35.24

## B.2 Usability

Table B.2 illustrates mean and standard deviation values of the SUS score and components of SUS scores associated for G1 and G2 with Vismep and Tracemalloc. SUS score varies from 0 (worst imaginable) to 100 (excellent). The average SUS score calculated from the participant’s answers for Vismep is 72.05 (std. dev. 12.62) for G1 and 73.64 (std. dev. 11.09) for G2. According to Sauro [125], Vismep is graded as “C+” and “B-”, which indicate a “good” usability score. The average SUS score recorded by participants using Tracemalloc is 44.09 (std. dev. 20.38) for G1 and 49.09 (std. dev. 21.22) for G2. These measures correspond to a grade of “F”, which indicates an “awful” usability score.

Table B.2: Means of overall SUS and components of SUS scores for G1 and G2 using Vismep and Tracemalloc.

	G1				G2			
	Vismep		Tracemalloc		Vismep		Tracemalloc	
	Mean	SD	Mean	SD	Mean	SD	Mean	SD
<b>Overall SUS</b>	72.05	12.64	44.09	20.38	73.64	11.09	49.09	21.22
<b>Usability aspects</b>								
Q1: Willing to use the tool	4	0.63	2.27	1.10	3.82	1.08	2.55	1.13
Q2: Complexity of the tool	1.91	0.83	3.09	1.04	2.27	1.01	3.27	1.42
Q3: Ease of use	4	0.45	3	1.18	3.91	0.94	3.27	1.01
Q4: Need of support to use	2.91	1.14	3.45	1.04	2.91	1.14	3.36	1.21
Q5: Integrity of functions	4.09	0.54	3.27	1.01	4	0.89	3.18	1.25
Q6: Inconsistency	1.82	0.75	2.91	0.94	1.36	0.67	2.45	0.69
Q7: Intuitiveness	3.45	1.21	2.36	1.43	4	1	2.63	1.29
Q8: Cumbersomeness to use	1.73	1.01	2.82	1.17	1.73	0.65	2.45	0.69
Q9: Feeling confident to use	3.82	0.60	2.18	0.60	3.82	0.98	2.55	1.04
Q10: Required learning-effort	2.18	1.17	3.18	1.33	1.82	0.75	3	1.26

**Components.** We detailed the SUS components' scores to understand the participant's perception 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. Regardless of the group, Vismep achieved higher scores for positive aspects and lower scores for negative aspects (except for the need for support to use Vismep). These results reflect that some programmers struggle with this tool due to (i) doubts about explicit information or interactions and (ii) its performance. Tracemalloc obtained some lower scores for positive aspects (willingness to use Tracemalloc, intuitiveness, and feeling confident to use) and some higher scores for negative aspects (complexity of Tracemalloc and need of support to use). These scores are related to the challenges and observations participants made since some perceived that obtaining the required information was challenging and dealt with errors when running the modified code to get the necessary information.