

Master Thesis

February 25, 2014

# Java Map Enabled Program Comprehension

**Marc Weber**

of Bern, Switzerland (02-800-076)

**supervised by**

Prof. Dr. Harald C. Gall

Dr. Emanuel Giger, Dr. Michael Würsch



University of  
Zurich<sup>UZH</sup>





Master Thesis

---

# Java Map Enabled Program Comprehension

Marc Weber



University of  
Zurich<sup>UZH</sup>



**Master Thesis**

**Author:** Marc Weber, <marc.a.weber@gmx.ch>

**Project period:** 01.09.2013 - 28.02.2014

Software Evolution & Architecture Lab  
Department of Informatics, University of Zurich



---

# Acknowledgements

I would like to thank Prof. Harald Gall for the opportunity to write this thesis at the s.e.a.l and for his trust in letting me work on the JAVA MAP once again. Many thanks to Emanuel Giger and Michael Wüsch for their suggestions and great support. I would also like to express my thanks to Beat Weisskopf, Sergio Trentini and Christian Lüthold for their critical, enriching comments. And finally many thanks to my family, especially to my dear wife Som, for her patience and support during these last six months.



---

# Abstract

Analyzing and understanding source code is one of the crucial tasks of every software developer. Object - oriented systems, with their logic distributed over several code entities are harder to understand and maintain than their procedural predecessors. But while modern software systems become more and more complex, software developers still have to use the same development tools already known for years. One of the main problems is the lack of a way to see the big picture.

To fill this gap, we present in this thesis the JAVA MAP. It is a set of tools, fully integrated into the JAVA DEVELOPMENT TOOLKIT (JDT) of ECLIPSE, with the aim, to support the developer in his everyday work. All tools are available at the developers finger tips, using state of the art software visualization techniques to ease program- as well as project history comprehension. The core is built by the JAVA MAP, a visual representation of the whole software system in focus. It is based on the concepts of the CLASS BLUEPRINT and allows the user to quickly grasp the overall picture of the elements in focus, as well as their interaction. This core is augmented by additional tools for type hierarchy understanding, as well as history analysis.

With a short user study at the end of our thesis, we have successfully verified the usefulness of the JAVA MAP in aiding program comprehension. The users of the study were significantly faster in solving the given program comprehension tasks when using the JAVA MAP.



---

# Zusammenfassung

Die Analyse und das Verstehen von Quellcode sind einige der wichtigsten Aufgaben eines jeden Entwicklers. Objektorientierte Systeme mit ihrer dezentralen Logik, verteilt über mehrere Codeelemente (Klassen), sind schwerer zu verstehen und zu warten als ihre prozeduralen Vorgänger. Doch während moderne Softwaresysteme immer komplexer werden, arbeiten die Entwickler seit Jahren mit den immer gleichen Werkzeugen. Eines der grossen Probleme ist die fehlende Möglichkeit, das Grosse Ganze zu sehen.

Um diese Lücke zu füllen präsentieren wir in dieser Arbeit die JAVA MAP. Sie ist eine Sammlung von Werkzeugen welche vollständig in das JAVA DEVELOPMENT TOOLKIT (JDT) von ECLIPSE integriert sind. Ihr Ziel ist es, den Entwickler bei seiner täglichen Arbeit zu unterstützen. Der Anwender kann auf alle Werkzeuge direkt zugreifen und es werden modernste Techniken aus dem Bereich der Software Visualisierung verwendet um sowohl das Programmverständnis als auch die Analyse der Projekthistorie zu erleichtern. Das Kernelement wird von der JAVA MAP gebildet, einer visuellen Darstellung des gesamten Softwaresystems im Fokus. Die Darstellung basiert auf der Idee der CLASS BLUEPRINT und erlaubt es dem Anwender, sich schnell und einfach einen Überblick über die fokussierten Elemente und deren Zusammenspiel zu verschaffen. Dieser Kern wird durch zusätzliche Werkzeuge zur Analyse der Typhierarchie und der Projekthistorie ergänzt.

Wir konnten mit einer kurzen Benutzerstudie am Ende unserer Arbeit erfolgreich aufzeigen, dass die JAVA MAP sich bestens eignet, um ein Softwaresystem zu analysieren. Die Benutzer der Studie waren signifikant schneller, beim Lösen von Aufgaben zum Programmverständnis, unter der Verwendung der JAVA MAP.



---

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Software Visualization Tools . . . . .	1
1.2	History . . . . .	2
<b>2</b>	<b>Related Work</b>	<b>3</b>
2.1	Categorization Framework . . . . .	3
2.2	Software Visualization Systems . . . . .	4
2.2.1	Class Blueprint . . . . .	4
2.2.2	Evolution Radar . . . . .	6
2.2.3	Code Swarm . . . . .	6
2.2.4	Code City . . . . .	9
<b>3</b>	<b>Java Map</b>	<b>11</b>
3.1	The Analyzer . . . . .	11
3.2	The Map . . . . .	12
3.2.1	Visual Elements . . . . .	12
3.2.2	Features . . . . .	16
3.3	Type Hierarchy View . . . . .	20
3.4	Timeline View . . . . .	23
3.5	Change Coupling View . . . . .	23
3.6	Tree Rings View . . . . .	24
<b>4</b>	<b>Implementation</b>	<b>27</b>
4.1	A Software Visualization Reference Model . . . . .	27
4.2	Component Architecture of the Java Map . . . . .	28
4.2.1	Container . . . . .	29
4.2.2	Analysis . . . . .	29
4.2.3	Persistence . . . . .	29
4.2.4	View . . . . .	30
<b>5</b>	<b>User Study</b>	<b>33</b>
5.1	Motivation . . . . .	33
5.2	Study Setup . . . . .	33
5.3	Tasks . . . . .	35
5.3.1	History (Coworkers) . . . . .	35
5.3.2	Type Hierarchy . . . . .	36
5.3.3	Entity Access . . . . .	37

---

5.3.4	Metrics . . . . .	39
5.4	Evaluation Results . . . . .	39
5.4.1	Overview . . . . .	40
5.4.2	Detailed Task Analysis and Interpretation . . . . .	41
5.5	User Feedback . . . . .	50
<b>6</b>	<b>Conclusion</b>	<b>53</b>
6.1	Conclusion . . . . .	53
6.2	Future Work . . . . .	53
6.2.1	Consolidating the Java Map . . . . .	53
6.2.2	Additional Features . . . . .	54
<b>A</b>	<b>Used Tools and frameworks</b>	<b>57</b>
<b>B</b>	<b>Contents of the CD-ROM</b>	<b>59</b>
<b>C</b>	<b>User Study</b>	<b>61</b>
C.1	Introduction Material . . . . .	61
C.2	Questionary . . . . .	68
C.3	Statistical Analysis . . . . .	77



## List of Figures

2.1	An example of a CLASS BLUEPRINT. (source: [DL05]) . . . . .	5
2.2	A CLASS BLUEPRINT created with the CBP-Plugin. (source: [Tre11]) . . . . .	5
2.3	Principles of the EVOLUTION RADAR. (source: [DLL09]) . . . . .	7
2.4	A simplified diagram of the CODE SWARM layout. (source: [OM09]) . . . . .	8
2.5	Vision of a revision radar of a user. . . . .	8
2.6	Visual representations of the CODE CITY. (source: [Wet09]) . . . . .	9
3.1	The context menu of the JAVA MAP. . . . .	11
3.2	Connection types of the JAVA MAP. . . . .	13
3.3	Components of the shape header. . . . .	15
3.4	Representation of a class in the JAVA MAP. . . . .	15
3.5	Representation of an interface in the JAVA MAP. . . . .	16
3.6	Representation of enumerations in the JAVA MAP. . . . .	17
3.7	Representation of packages in the JAVA MAP. . . . .	17
3.8	Example of a project with one package root. . . . .	18
3.9	Examples of selections. . . . .	19
3.10	Hovering over the method <i>tearDown</i> . . . . .	19
3.11	The outline view of the JAVA MAP. . . . .	20
3.12	The expand/collapse mechanism using the example of the class <i>IvyConvertPom</i> . . .	21
3.13	A shape header of a class which is part in an inheritance hierarchy. . . . .	21
3.14	Example of a Type Hierarchy View. . . . .	22
3.15	Example of the Timeline View of the JAVA MAP. . . . .	23
3.16	Example of the Change Coupling View. . . . .	24
3.17	Example of the Tree Rings View of the JAVA MAP. . . . .	25
3.18	The selection mechanisms of the Tree Rings View. . . . .	26
4.1	A Reference Model for Software Visualization. (source: [MMC02]) . . . . .	27
4.2	The basic components of the JAVA MAP. . . . .	28
4.3	Illustration of the versioning mechanism of the JAVA MAP persistence. . . . .	30
4.4	The persistence model of the JAVA MAP. . . . .	31
5.1	Box plot of the cumulated completion times. . . . .	41
5.2	Box plot for Task H.1.1. . . . .	43
5.3	Box plot for Task H.1.2. . . . .	43
5.4	Box plot for Task T.2.1. . . . .	44
5.5	Box plot for Task T.2.2. . . . .	45
5.6	Box plot for Task T.2.3. . . . .	46
5.7	Box plot for Task E.3.1. . . . .	47
5.8	Box plot for Task E.3.2 . . . . .	47
5.9	Box plot for Task E.3.3. . . . .	48
5.10	Box plot for Task M.4.1. . . . .	49
5.11	Box plot for Task M.4.2. . . . .	50
5.12	Box plot for Task M.4.3. . . . .	51

## List of Tables

2.1	Attribute summary for the analyzed systems along the five dimensions. . . . .	10
3.1	Color code used for the methods in the JAVA MAP. . . . .	14
5.1	Test procedures for the different test groups. . . . .	35
5.2	The questions of the history block. . . . .	36
5.3	The questions of the type hierarchy block. . . . .	37
5.4	The questions of the entity access block. . . . .	38
5.5	The questions of the metrics block. . . . .	40
5.6	Test Hypotheses. . . . .	40
5.7	Summary of the individual results per task. . . . .	42

# Introduction

Reading source code and building an understanding of the underlying system is the bread-and-butter activity of every developer. However, analyzing the code base of any real-world, productive system is time consuming and imposes numerous challenges, such as code written by various other developers or understanding the distributed logic of object-oriented software. Specific scientific communities, such as ICPC<sup>1</sup> or VISSOFT<sup>2</sup>, dedicated themselves to developing advanced and effective methods to support program comprehension. In addition, the field of software visualization employs principles of scientific visualization to facilitate code understanding. The need for support in program comprehension is mainly time and cost driven. Already in the year 2000, maintenance claimed between 85% and 90% of the overall costs of an information systems [Erl00]. If we combine this with the fact that over half of the time spent by a developer is dedicated to program comprehension tasks [ZSG79]. This implies a huge potential for any type of tool support.

## 1.1 Software Visualization Tools

Today, there exists a multiplicity of tools, which support the daily work of a developer. Examples are automated build systems, Integrated Development Environments (IDEs), runtime analyzing tools to investigate the consumption of memory and processor cycles, metric tools which generate statistical reports, test environments and much more. Among all these tools a special group is built by the software visualization tools for a simple yet powerful reason. While non-visual tools simply do their work and present the results to the user, software visualization tools focus on the synergy produced between the user and the tool during the usage. Their aim is to utilize the capabilities of the human visual system (HVS) to do parts of the overall work. For example, the HVS allows a person to quickly and without much effort, find a single red circle in a large group of black boxes. While this is absolutely natural for a human, the same task can be quite complex, time consuming, and even error-prone if solved by a machine. Therefore, the main idea behind most visualization tools is to preprocess the information and then present it to the user for the final step of the analysis, like inference and reasoning. The tool does not need to know or interpret anything. All that is needed, is to represent the found result to the user, which will quickly find the elements of interest by himself. The overall strength of the tool depends on the visual representation and how easy it is for the user to access the needed information. The smaller the gap between the picture presented on screen and the mental model in the head of the user is, the higher the synergy and the more useful the tool becomes.

---

<sup>1</sup><http://www.program-comprehension.org>

<sup>2</sup><http://icsm2013.tue.nl/VISSOFT/>

## 1.2 History

Although just a short one, the JAVA MAP already has a history. It sprung in to life in 2011 as a prove of concept described in [Web11]. During the master project from 2012 - 2013 [Web13], the code of the map was refactored and a persistence layer, as well as an access to the SOFAS<sup>3</sup> was integrated to the tool. Finally, at the end of 2013 we started with this master thesis to finalize the tool by adding history information, as well as new visualizations.

### The goal of this thesis

The main goal of this master thesis was to finalize the JAVA MAP in a way that it becomes a useful aid for program comprehension. Therefore, a data extraction service targeting GIT repositories as well as new visualizations were added to the code base. To verify if our endeavor was successful, we conducted a user study towards the end of our thesis.

---

<sup>3</sup><http://www.ifi.uzh.ch/seal/research/tools/sofas.html>

# Related Work

This chapter presents a brief overview over some tools and approaches from the domain of software visualization. To support the tasks of analyzing the different approaches and comparing them to each other, as well as to the JAVA MAP later on, we use the framework proposed by Maletic et al. to classify them. The framework is explained in section 2.1. The different software visualization systems are presented in section 2.2. For each system, we give a short overview over their functionality, strengths and weaknesses and explain if and how they influenced the design and functionality of the JAVA MAP. A summary of the different systems presented is given in the Table 2.1 at the end of this chapter.

## 2.1 Categorization Framework

The framework is based on the work of Price [PBS93] and Roman [RC93] and refined to particularly *"emphasize the general tasks of understanding and analysis during development and maintenance of large-scale software systems."* [MMC02]

The framework categorizes software visualization systems along the five dimensions:

**Task** - *Why* is the visualization needed?

Typically visualization systems support and foster the understanding of different attributes of a software system. This understanding can be used to solve certain tasks. Different tasks need different information and may therefore be supported by different visualization tools. In other words, the task dictates the bounds of the information presented, as well as the visual structure (graphical representation) to some extent. In the view of the authors, *"this dimension is the driving force behind defining a classification of software visualization systems. If such a system does not support the engineering task on the user's agenda, the other features are of no importance."* [MMC02] Furthermore, this dimension may indirectly influence the audience dimension as well, because some tasks are executed mainly by one group of users, like developers, architects, managers, etc.

**Audience** - *Who* will use the visualization?

This dimension defines the user group(s) the analyzed software visualization is targeting. Examples are academia (educational purpose), industry, developers, testers, managers, etc., as well as a mixtures of these groups. Also included in this dimension is the skill level of the users: Experts versus newcomers. The higher the skill level, the more complex the tool can be. But on the other hand the steeper the learning curve for a newcomer is.

**Target** - *What* is the data source to represent?

The target can be the raw data sources like code, metrics, documentation, history information, etc., as well as derived information from these sources like architecture, design, business process, etc. In general, it stands for the information presented to the user. This dimension influences the precision, as well as the number of supported tasks by the tool. If a tool supports many targets, the amount of information is too big to be presented as a whole to the user. Therefore abstraction and aggregation is needed to chop down the amount of information, at the cost of precision. The benefit of this approach is of course the capability to support many different tasks. On the other side, a tool with a small and precise target can give the user much higher insights and thereby support a certain task better than the general tool, but on the downside may only support a small number of specialized tasks. Last but not least this dimension also focuses on the scalability issue: It is important that a visualization tool is not dependent on the size of the target, but rather capable to handle and present small, as well as large targets alike.

#### **Representation- How to represent it?**

This dimension focuses on the capabilities of the visualization tool to map the underlying target to a visual representation. The two criteria *expressiveness* and *effectiveness* defined by MacKinlay [Mac86] can be used to evaluate these mapping capabilities. As explained by Maletic et al., "*expressiveness refers to the capability of the metaphor to visually represent all the information we desire to visualize*", while "*effectiveness, relates to the efficacy of the metaphor as a means of representing the information. [...] Effectiveness implies the categorization of the visual parameters according to its capabilities of encoding the different types of information. Moreover, this also implies categorizing the information according to its importance so that information that is more important can be encoded more efficiently when options must be taken.*" [MMC02] Furthermore, the authors propose to evaluate the visualization tool according to the seven high-level user needs defined by Shneiderman [Shn96] which are *Overview, Zoom, Filter, Details-on-demand, Relate, History and Extract*.

#### **Medium - Where to represent the visualization?**

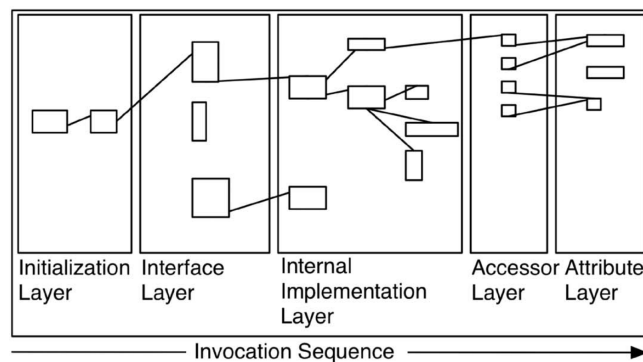
This dimension focuses on the physical presentation device, like a piece of paper, a computer screen, a beamer, etc. The authors solely focus on the physical devices possible, which in most cases is a computer screen. Because this criteria does not give us much room to distinguish the different tools we analyze, we like to redefine this dimension in the following way: We extend the definition of '*where*' to include as well, where the tool appears or is used in the context of the working processes. So '*where*' additionally answers questions like '*Does the tool stand alone or is it integrated for example in a IDE or a build process?*' or '*From where can the tool be accessed?*'.

In the next section we present several software visualization systems and explain their features along these five dimensions. A summary is given at the end in Table 2.1.

## **2.2 Software Visualization Systems**

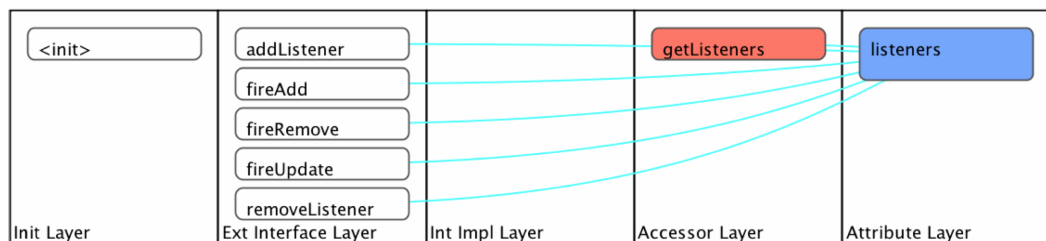
### **2.2.1 Class Blueprint**

The CLASS BLUEPRINT is a 2D graphical representation of a single class and its children (methods and fields). The aim of the visualization is "*to ease the understanding of classes by visualizing a semantically augmented call and access-graph of the methods and attributes of classes.*" [DL05] A CLASS BLUEPRINT is a simple rectangle box which is subdivided into five layers as shown in Figure 2.1. The child elements of the class are placed in the according layer. For example, fields are placed in the attribute layer, getter and setter methods fall into the access layer, private methods are placed in the internal implementation layer, and so forth. The layout of the elements of a



**Figure 2.1:** An example of a CLASS BLUEPRINT. (source: [DL05])

layer represent use or call relations. "The layers support a call-graph notion in the sense that a method node on the left connected with another node on the right is either invoking or accessing the node on the right that represents a method or an attribute." [DL05] In addition, the concept of polymetric views [LM06] is used to map different metric values to the shapes representing the source code elements. Up to three metrics can simultaneously be mapped to each element (method or attribute), one on the height, one on the width and one on the color (e.g., the darker the color the higher the value). The color itself can be used to distinguish different types of element, e.g., public and protected methods. Initially, the CLASS BLUEPRINT was designed to analyze smalltalk classes, later a standalone version supporting Java was created. In 2011 a version directly integrated into the Eclipse IDE called "CBP-Plugin" was created by Trentini [Tre11]. Figure 2.2 shows a CLASS BLUEPRINT created by the CBP-Plugin. The CLASS BLUEPRINT offers a basic overview of the



**Figure 2.2:** A CLASS BLUEPRINT created with the CBP-Plugin. (source: [Tre11])

internals of a class. One of its biggest advantages is the capability to always fit the available space: By using the same scaling factor for all elements, the blueprint can be shrunk to fit any given bounds, be it a certain paper size or a specific resolution of a monitor or beamer. Because all parts of the visualization are scaled by the same proportion the overall picture and therefore the semantics stay the same! Furthermore, over time the user is capable to recognize recurring visual patterns. As described by Lanza et al., these visual patterns help the user to even faster grasp the essentials of a class and its structure.

## Influence of the Class Blueprint on the Java Map

The CLASS BLUEPRINT had a major influence on the JAVA MAP. It built the starting point for the main editor and most of the concepts were inspired from it. One could say that the JAVA MAP is basically the next generation of the CLASS BLUEPRINT.

### 2.2.2 Evolution Radar

The EVOLUTION RADAR is a tool which completely focuses on CHANGE COUPLING. Change couplings is based on the idea of logical coupling which is *"the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system."* [GHJ98] In other words, if two files or entities have changed together several times in the past, there is a high possibility that these two elements have some kind of a logical relationship, one is dependent on the other. Therefore, if we trace files which frequently change together, we will most likely find these otherwise hidden relationships. The difference between change and logical coupling seems small, but is rather important: Change coupling is a simple measurement of how often artifacts change together, while logical coupling is a measurement of how strong two artifacts are connected logically. While the first can be automatically calculated, the latter needs reasoning and therefore human intervention. Therefore, the EVOLUTION RADAR focuses on change couplings which can easily be calculated and showed to the user. With the aid of the radar, the user is then able to find the logical couplings in the system.

As the name indicates, the EVOLUTION RADAR presents the information in a circular representation to the user. The principles of the visual representation are shown in figure 2.3. *"The EVOLUTION RADAR shows the dependencies between a module in focus and all the other modules of a system. The module in focus is represented as a circle and placed in the center of a circular surface. All of the other modules are visualized as sectors, whose size is proportional to the number of files contained in the corresponding module. The sectors are sorted according to this size metric and placed in clockwise order. Within each module sector, files belonging to that module are represented as colored circles and positioned using polar coordinates."* [DLL09] The distance  $d$  indicates the strength of the coupling: The more coupled the file is to the module in focus, the smaller  $d$  is. The angle assigned to the file is based on the alphabetical ordering of the full directory path, thereby files in the same package are close together.

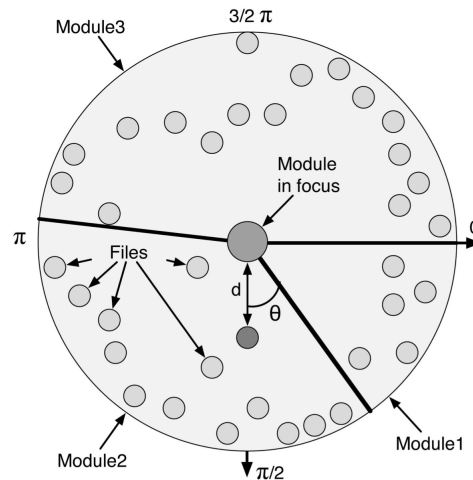
## Influence of the Evolution Radar on the Java Map

The EVOLUTION RADAR served as a direct template for the Change Coupling View of the JAVA MAP.

### 2.2.3 Code Swarm

CODE SWARM is a free open source application. The basic concept is to use an organic information visualization to generate a video of the history of a project. The tool is focused on appealing graphics therefore, no exact quantities and relationships between the elements of the graph are used. The reason for this decision is explained by the authors as follows: *"Organic information visualization is an inherently fuzzy method of data display: there are no exact quantities or relationships being shown. However, it is an acceptable level of ambiguity because our goal is not hard analytics. As long as people get the impression of what is happening in the project, exact quantities are not needed."* [OM09] Because of the simple and appealing graphics, CODE SWARM addresses developers, as well as casual viewers alike. A schematic of the generated visualization is shown in Figure 2.4. The mean-





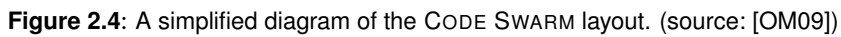
**Figure 2.3:** Principles of the EVOLUTION RADAR. (source: [DLL09])

ing of the different elements is as follows: "(A) Colored labels indicate the file type. (B) Document files, as blue circles, have been committed by "documenter." The dark color means they were committed close to the current time. (C) Source code files, as red circles, have been committed by "programmer" and "helper." Some circles are lighter, which means they were committed earlier than the darker ones. (D) A histogram tracks the amount and type of commits over time, from right (newer) to left (older). (E) The date display provides temporal context during the animation." [OM09] The alignment of the authors and files is done through a simple spring embedder algorithm: "When a developer commits a file, an [invisible] edge is created between developer and file nodes and they attract each other. As time passes, the attractive force of the edge weakens and they move apart. Files also repulse other files so that there is not so much overlap between unrelated ones. Developers, however, neither attract nor repel other developers. This means that they are positioned by the files alone. Therefore, two spatially close developers work on the same files." [OM09]

Although the lack of exact quantities may hamper the usefulness for developers in their daily work, the decision was right for the task the authors wanted to achieve. Reaching a large group of viewers, both from the professional development area, as well as casual viewers. Finally a community was formed around CODE SWARM, the application was made open source and because of various contributions today most major version control systems like CVS, SVN, GIT and Mercurial are support.

### Influence of the CODE SWARM on the JAVA Map

While the basic idea of including something similar to the CODE SWARM into the JAVA MAP is appealing, we saw two main problems. First, the lack of accuracy. We need a precise and unambiguous representation if the visualization should be of any use other than pure entertainment. Second, the representation of time. Videos or long running animations are no option, we rather need a static representation, which includes the additional dimension of time. Inspired by the layout algorithm of the swarm, which puts files recently changed by a user closer to its name and



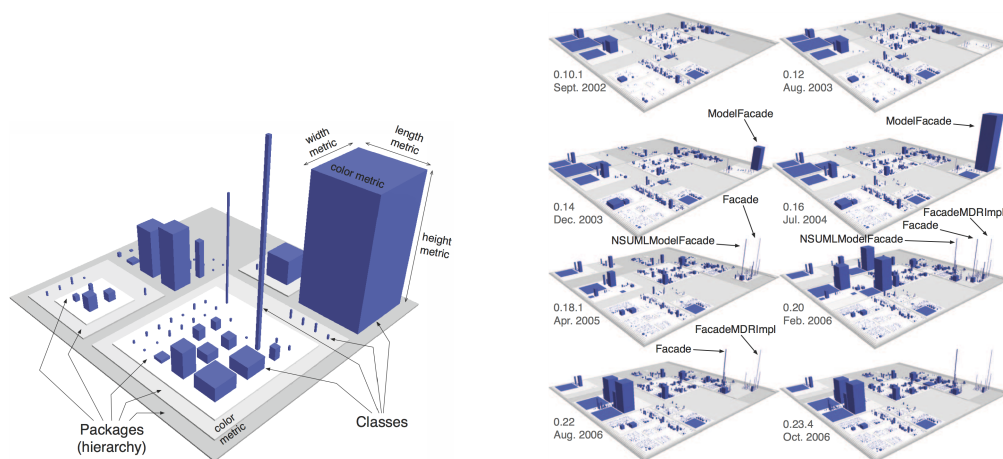
**Figure 2.5:** Vision of a revision radar of a user.

### 2.2.4 Code City

CODE CITY is a 3D visualization which uses the metaphor of a city to represent the elements of a software system. An example is shown in Figure 2.6(a). Packages are represented as districts, which contain other districts, as well as buildings. In this way, *"the package hierarchy is reflected by the city's district structure."* [Wet09] The buildings represent the classes of the system. The concept of polymetric views is used to map metrics to the visual properties of the city artifacts. The default configuration maps for classes the number of methods (NOM) to the buildings height and the number of attributes (NOA) to the buildings base size. For the districts (packages) a color scheme on the saturation level is used to indicate the nesting level.

The great advantage of the CODE CITY compared to other visualization approaches, is the ability to show a complete system in a single visual representation. Through scaling, even large systems can be visualized on a limited space, like a paper or a computer screen. The mapped metrics lead to several different visual patterns depending on the height and base size of a building. To better distinguish different types of classes, Wetzel and Lanza [WL07] defined five types of buildings with according thresholds for their sizes: *House*, *Mansion*, *Apartment Block*, *Office Building* and *Skyscraper*. Wetzel added a sixth one, the *Parking Lots* [Wet09], which represent classes with few methods and many attributes (data containers). These visual classifications allow a user to identify outlines, like classes with many methods (Skyscrapers), or monoliths containing a huge chunk of logic and fields (Office Buildings), quickly and without much effort.

While analyzing a project, just focusing on the current state of the system is often not enough and can result in misleading conclusions. Therefore, the CODE CITY provides a feature called *Time Travel* which allows the user to analyze the system over a period of time as shown in Figure 2.6(b). As explained by Wetzel, *"time travel allows stepping through the versions of the system and observing the changes inside the city. To enable such observations we ensure consistent locality, i.e., each artifact representing a software entity is assigned a lifetime real-estate in the city. The empty spaces left behind by the removal of entities are never reallocated."* [Wet09]



(a) A basic visualization of a system with CODE CITY.

(b) Example of a CODE CITY time travel through the history of the project ArgoUml.

**Figure 2.6:** Visual representations of the CODE CITY. (source: [Wet09])

## Influence of the Code City on the Java Map

The influence of the CODE CITY on the JAVA MAP was rather limited. First of all, the CODE CITY is a 3D approach, while all views and editors of the JAVA MAP are 2D based. However, the idea of time travels in the visual representation fascinated us and we tried to come up with a solution which fits into the Map. Unfortunately, there is no easy solution to this because of two reasons. First, the interactive nature of the JAVA MAP. Only the focused elements of a system are fully expanded, while the rest is shrunk as much as possible while still preserving the overall context. Second, the Map is organized based on the call dependencies of the elements. As these change over time, we would have no stable locality of the elements which renders an interactive approach useless.

<b>Dimension</b> <b>Tool</b>	<b>Task</b>	<b>Audience</b>	<b>Target</b>	<b>Representation</b>	<b>Medium</b>
<b>Class Blueprint</b>	Reverse engineering, class analysis, maintenance	Expert developer	Class files	2D graphic, polymetric views, flow structures, interactive(CBP-Plugin), independent of available size	Color monitor, standalone, integrated in Eclipse IDE
<b>Evolution Radar</b>	Dependency analysis, maintenance	Expert developer	History information, change coupling	Interactive, 2D graphic, circular radar diagram, color, relative change coupling between focus module and rest of system	Color monitor, standalone
<b>Code Swarm</b>	Overview of history, entertainment	Broad audience	History information	2D video, organic graph layout	Color monitor, standalone, video
<b>Code City</b>	Reverse engineering, system analysis, maintenance	Expert developer	Source code, nesting structures, project history	3D graph, interactive, city metaphor	Color monitor, standalone

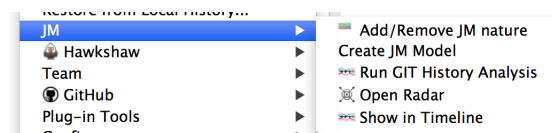
**Table 2.1:** Attribute summary for the analyzed systems along the five dimensions.

# Java Map

In this Chapter the JAVA MAP is presented from a user's perspective, while technical details are explained in Chapter 4. The main goal of the JAVA MAP is to support the user in different scenarios by offering additional information, if needed. The map is designed as an extension to the normal ECLIPSE development environment, by adding different editors and views. In the following sections, we go through each of these elements, explain their features and highlight how they can support the user in different situations. Section 3.1 introduces the different analyzers and how to use them. In Section 3.2 the main editor is presented. The Type Hierarchy View is explained in Section 3.3. Finally, the history related tools, like the Timeline View, the Change Couplings View and the Tree Rings View, are presented in the Sections 3.4, 3.5 and 3.6 respectively.

### 3.1 The Analyzer

In order to use the different functionalities provided by the JAVA MAP, the user first needs to run the according analysis of the Map. To start the analysis, the user selects the Java project of interest in the Package Explorer and opens the context menu. The commands for the JAVA MAP can be found in the submenu *JM*, as depicted in Figure 3.1. The JAVA MAP provides three types of analyses. The current version of the project can be analyzed with the command *Create JM Model*. Once the analysis is completed, the JAVA MAP can be accessed. If the project is attached to a GIT repository, the JAVA MAP can analyze its history through the command *Run GIT History Analysis*. After the history analysis is finished, features like timeline and Tree Rings View become available as well. Finally, if the user wants to keep the JAVA MAP up to date, he can add the JAVA MAP Nature to the project through the command *Add/Remove JM nature*. This causes the JAVA MAP to analyze the changes made to the local project and update its database accordingly.



**Figure 3.1:** The context menu of the JAVA MAP.

## 3.2 The Map

The main contribution is an editor called after the whole tool: the JAVA MAP. It is based on the ideas of the CLASS BLUEPRINT introduced by Lanza et al. [DL05]. The underlying source code elements are represented as simple boxes. The main goal is to represent the elements of a system (code base) in a simplified and abstracted way to get an easier understanding of the components and their relations. Containment relations (e.g., methods in a class) are depicted as boxes inside their parent box and the elements of a container are laid out according to their relation among each other. The concept of polymetric views, as described in [LM06], is used to map different properties to the size and color of the boxes. In contrast to the CLASS BLUEPRINT, the JAVA MAP not just shows one class at the time but includes the whole context as well (e.g., the currently analyzed project). Furthermore, the JAVA MAP is dynamic: The user can move from the source code directly into the Map and jump back into the code if needed. The Map can be zoomed and container elements can be opened and collapsed. In Section 3.2.1 we present each code element in detail, by explaining its visual representation and layout. The different features of the Map are explained in Section 3.2.2 and a special extension of the Map, the *Type Hierarchy View*, is explained in Section 3.3

### 3.2.1 Visual Elements

In this section we present the representation details for every component of the JAVA MAP. We explain how the map depicts e.g., classes and interfaces and how the whole package hierarchies are represented.

#### Layout

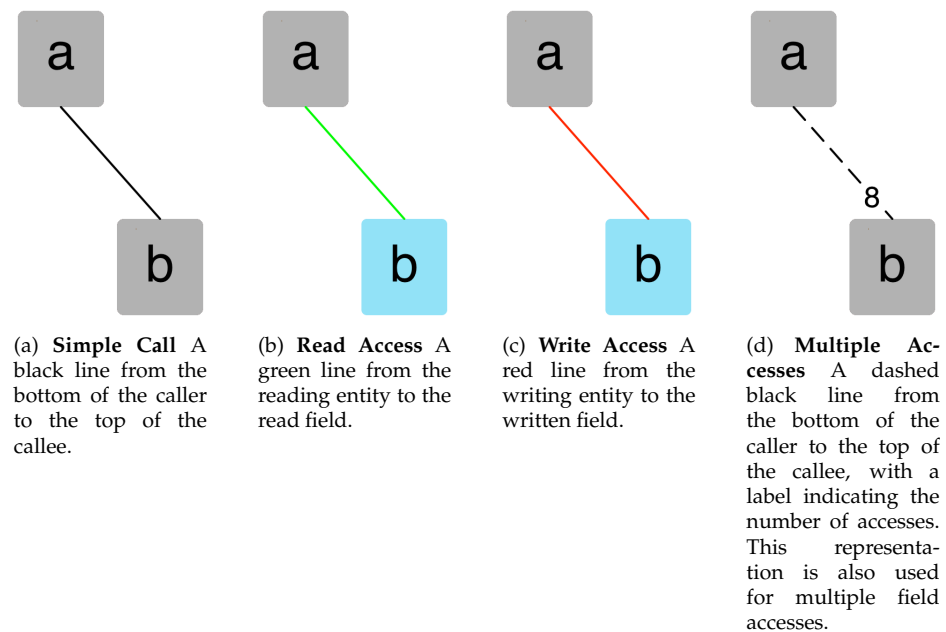
The JAVA MAP uses two simple rules to lay out the elements:

1. The shapes are nested based on their parent-child relations. Contained elements (e.g., methods in classes) are placed inside of their parent.
2. Elements in a container are laid out based on the call relation (method access) between the source code elements by following the simple rule '*a called shape is positioned under its caller*'. This holds for all elements on any level. Examples are methods in classes, classifiers in packages and packages in their super package or package root, etc.

The call relation is represented by a line starting at the bottom of the caller and ending at the top of the callee, as depicted in Figure 3.2(a).

#### Connections

Relations between elements are represented as lines. The JAVA MAP distinguishes between four types of relations, as depicted in Figure 3.2.



**Figure 3.2:** Connection types of the JAVA MAP.

## Simple Shapes

The simple or basic shapes in the JAVA MAP represent the smallest entities.

### Fields

The fields of a classifier are depicted as a simple boxes with an icon indicating the access modifier (e.g., public, protected, etc.). The JAVA MAP distinguishes two types of fields:



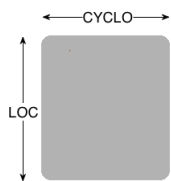
**Simple Field** - It corresponds to the simple types in Java (e.g., int, char, boolean, etc.) and is used to represent *enumeration constants* as well.



**Type based Field** - All object based fields, which directly or indirectly inherit *Object*.

### Methods

Methods are simple boxes with a certain width, height and color:



The height indicates how many lines of code (LOC) this method consists of, while the width shows the cyclomatic complexity (CYCLO), which is the branching factor (number of if/else and switch statements). Tall methods consist of many lines of code and take therefore more time to read, while broad methods have several execution paths and may therefore take more time to understand.

The color indicates the inheritance to the super method if one exists. The meaning of the different colors is shown in Table 3.1



**Normal method** - Defined in a class.



**Declaring method** - Declarations in interfaces and abstract classes. These methods have no body.



**Implementing method** - These methods implement a definition of an interface or abstract superclass.



**Extending method** - A method which extends the behavior defined by its super method (containing a super call in its body).



**Overwriting method** - A method replacing the behavior of its super method.

**Table 3.1:** Color code used for the methods in the JAVA MAP.

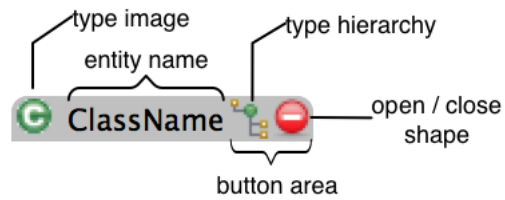
## Container Shapes

All entities which contains other entities (e.g., a class contains methods and fields) are called container shapes. They are represented by boxes having a header and several layers. A typical shape header is shown in Figure 3.3. It consists of an image indicating its type (e.g., class, interface, etc.), the name of the entity and a button area to the right side of the header. The button area contains controls to get more information for the corresponding shape. The rightmost button is always available. It is used to open or close the corresponding shape. The type hierarchy button is optional and only present if the underlying classifier is part of an inheritance hierarchy (e.g., a class having a superclass or implementing an interface).

### Class

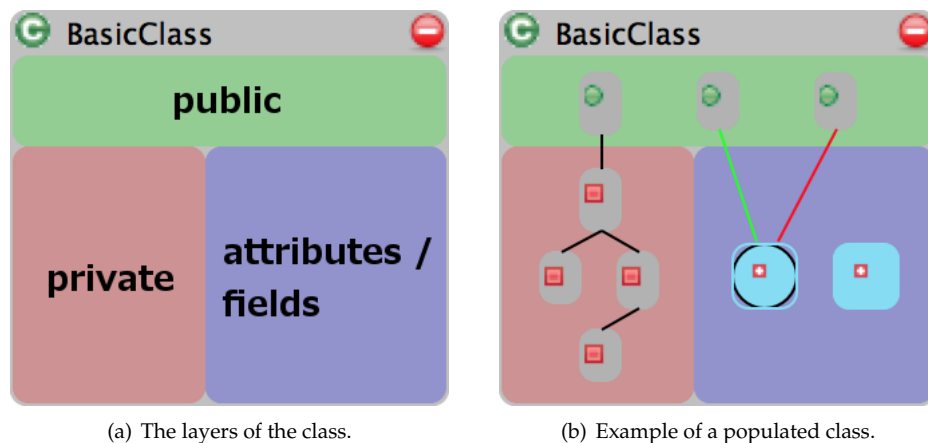
Classes consist of three layers, as depicted in Figure 3.4(a). The green layer at the top contains all public, protected and package private methods and is called the *access layer*. It is placed at the top, because all calls to this class are coming from above. The reason for this is the layout mechanism as described in Section 3.2.1. The red layer on the bottom left contains all private methods and can be seen as the container of the classes internal mechanics or business logic. The purple layer on the bottom right contains all attributes (fields) and inner classifiers (classes, interfaces and enumerations), if any. The size of the layers depend on the number of contained element and their relations to each other. Because the public and private layer use the layout described





**Figure 3.3:** Components of the shape header.

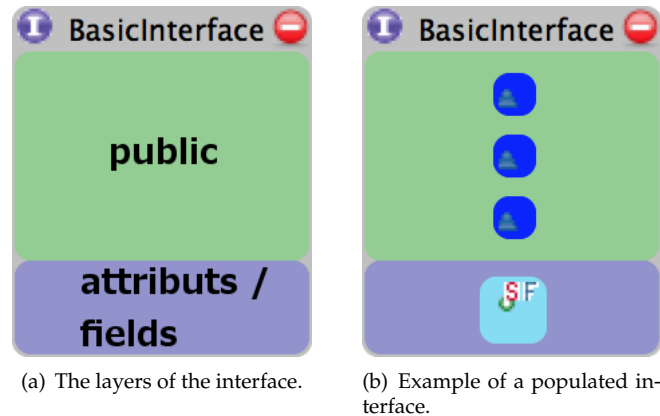
in Section 3.2.1, visual patterns emerge. The taller a layer the more call levels it contains. Therefore, the height of a layer is an indication of the call complexity of the elements in this layer. The class depicted in Figure 3.4(b), f.e., has a simple public layer and a more complex private one, containing three call levels. We can immediately see that at least one execution path in the private layer includes three methods, while another one consists of just two. We also see a simple entry point to the internal logic (public method on the left) and that the private fields of the class are not used by any private methods. Without reading a single line of code, we already have a basic understanding of the elements and their relationships in the given class.



**Figure 3.4:** Representation of a class in the JAVA MAP.

### Interface

Interfaces consist of two layers, as depicted in Figure 3.5(a). The green top layer contains the method definitions, while the purple bottom layer holds fields defined in the interface. The elements in both layers are laid out vertically. An example of a populated interface is shown in Figure 3.5(b).



**Figure 3.5:** Representation of an interface in the JAVA MAP.

### Enumeration

Enumerations are represented by two slightly different representations, based on their internal structure. Simple enumerations, which only hold constant values, consist of a single layer as depicted in Figure 3.6(a). The constants are laid out vertically. Complex enumerations, containing methods and fields as well, are depicted with a representation similar to the one of classes. An example is shown in Figure 3.6(b).

### Package

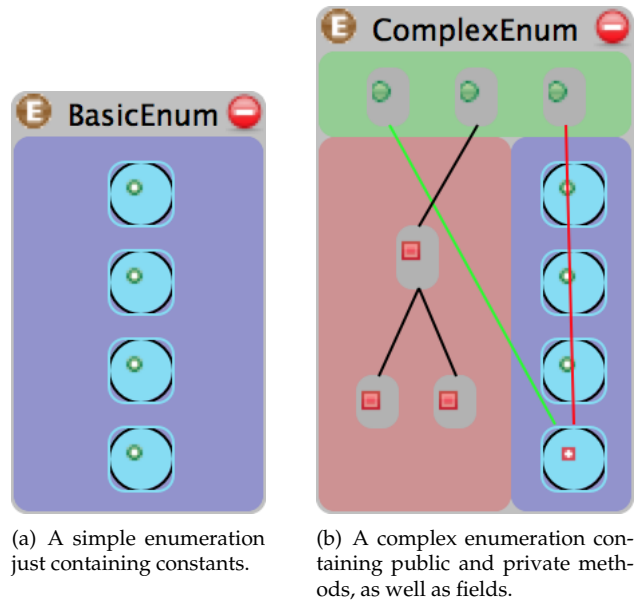
Packages consist of two layers. Classifiers (classes, interfaces and enumerations) are placed in the left layer, while the right layer contains all subpackages. Both layers are laid out with the layout described in Section 3.2.1. An example of a package is shown in Figure 3.7

### Project & Package Root

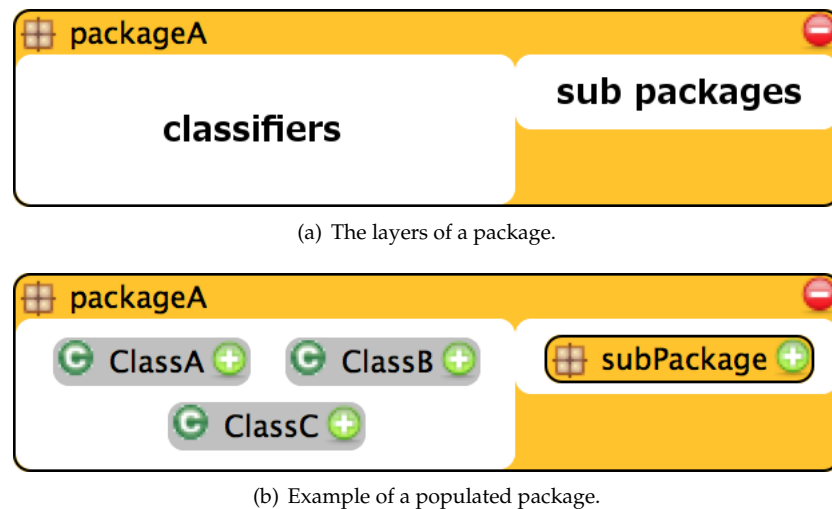
Projects build the top level elements in the JAVA MAP. They consist of only one layer which holds the package roots of the project. The package roots again only have one layer containing their packages. An example of a project with one package root which contains two packages is shown in Figure 3.8. All layers, as well as the projects themselves, are laid out using the layout described in Section 3.2.1.

## 3.2.2 Features

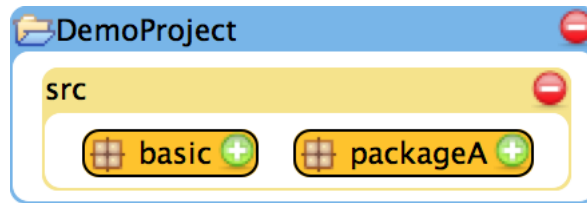
In this section we present the different features provided by the JAVA MAP for both, navigation, as well as details inspection.



**Figure 3.6:** Representation of enumerations in the JAVA MAP.



**Figure 3.7:** Representation of packages in the JAVA MAP.



**Figure 3.8:** Example of a project with one package root.

## Switching Between Source Code and Java Map

The switching mechanism is based on the idea that the JAVA MAP extends the basic Java source code editor of ECLIPSE, by providing a tool to zoom out of the code into a map like representation of the source code elements. To zoom out, the user places the cursor over the element of interest (e.g., a class name, a field, a method name, etc.) and presses the key combination **CTRL + "arrow down"**. This causes the editor to shrink into the element of the map which represents the current source location (cursor position). To move back from the JAVA MAP to the source code, the user simply double clicks on the element (shape) of interest. The view is automatically moved to the corresponding source code location in the editor. In case no Java editor is opened for the selected element, an editor is opened before the navigation starts.

## Zoom

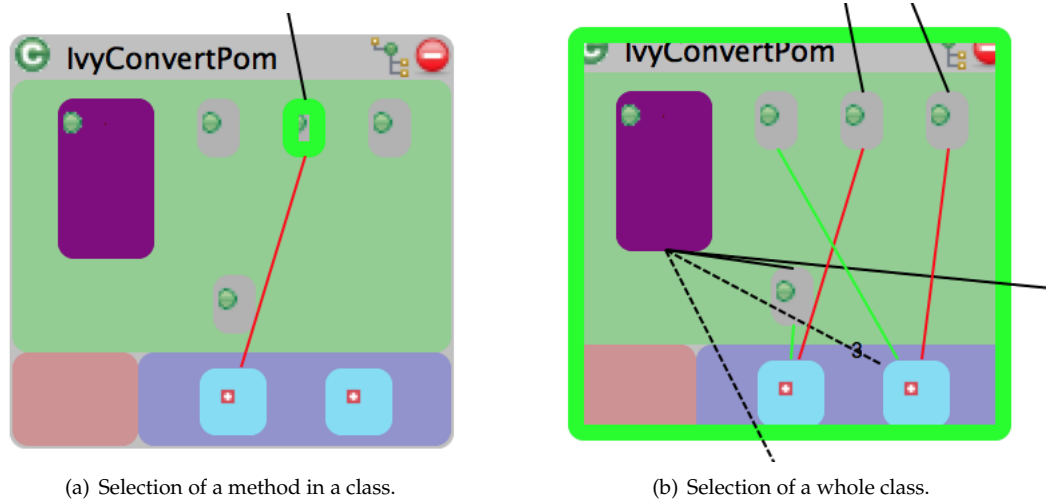
A simple zoom function allows the user to shrink or enlarge the Map as needed. The zoom is initialized by holding down the **CTRL** - key while using the mouse wheel. Beside the simple scaling, the zoom also repositions the visible area of the editor. The resized map is centered around the element underneath the mouse pointer. Therefore, the zoom can be used for navigation purposes in the following way. The user zooms out of the map, then he positions the mouse over the element of interest and zooms in again. With this technique the user is able to navigate quickly through the Map, even if the analyzed system is rather big.

## Selection

The JAVA MAP provides a selection mechanism to highlight elements of interest. A selected element is represented by a thick green border. Multiple selection is possible as well by holding down the **CTRL**-key while clicking on several elements. For a selected shape, all incoming and outgoing calls are shown. Figure 3.9(a) shows an example of a selected method in a class. If a containing shape, like for example a class is selected, the calls from and to all its children are shown in the map (see Figure 3.9(b)). Furthermore, the metrics of the selected element are shown in the info tab of the outline view.

## Hovering

Hovering (holding the mouse pointer still for a short time) over an element of the Map will bring up a small tooltip showing the image and name of the corresponding element. Figure 3.10 shows an example where the mouse is positioned over the method *tearDown* of the class *TestPerformance*.



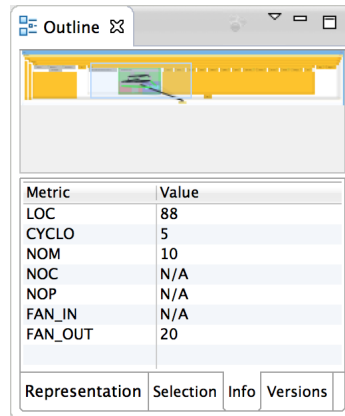
**Figure 3.9:** Examples of selections.



**Figure 3.10:** Hovering over the method *tearDown*.

## Outline View

Because the JAVA MAP is an ECLIPSE editor, it is equipped with an own outline view as shown in Figure 3.11. The top part consists of a thumbnail of the current editor. The blue rectangle indicates the current view location of the Map shown. The user can drag the rectangle to adjust the displayed part of the Map. The lower part of the outline view contains a table showing the metrics of the currently selected element of the map.



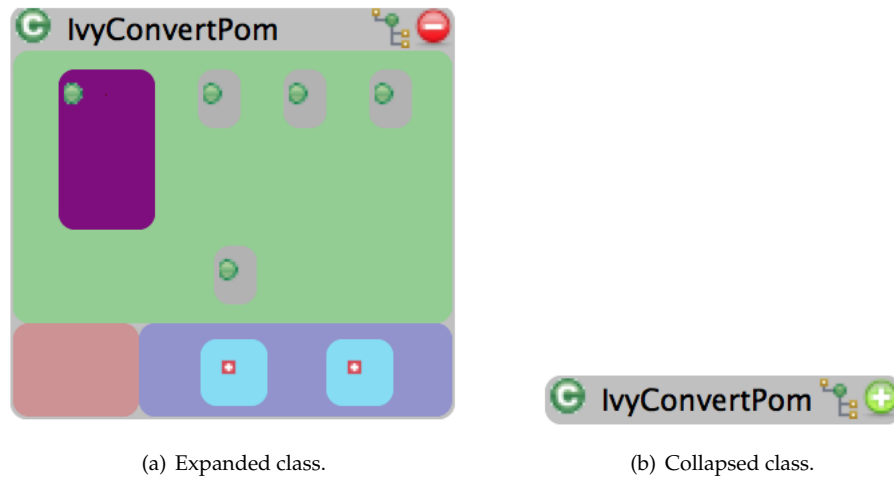
**Figure 3.11:** The outline view of the JAVA MAP.

## Collapsing and Expanding Shapes

These two features are available for all containing shapes of the JAVA MAP, like classifier, package, project, etc. A button in the top right corner of the element allows to open and close the specific shape. Figure 3.12(a) shows the opened class *IvyConvertPom* with the red close button on the right end of the shape header. When clicking the button, a short animation shrinks the class and only the shape header remains, as depicted in Figure 3.12(b). Please note that the sign of the button has changed to a green plus sign indicating that the shape can be opened by clicking on it. The collapse feature can be used to shrink uninteresting parts of the Map to a minimal representation, while the expand feature supports deeper investigation by allowing the user to drill down into interesting sections of the Map.

## 3.3 Type Hierarchy View

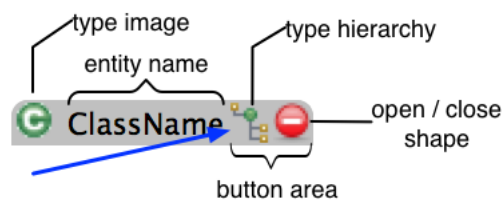
To analyze the inheritance relations between classes and interfaces of a given project, the JAVA MAP offers the Type Hierarchy View. It is accessible directly from the Map. If a shape is part of an inheritance relation, the Type Hierarchy button is shown in the shape header as depicted in Figure 3.13. Clicking on this button causes the JAVA MAP to generate and open a type hierarchy for the selected shape. An example is shown in Figure 3.14. In this view, the selected classifier (interfaces or classes), together with all super and sub classifier is shown. The green dashed arrows indicate the inheritance relation between classifiers, while the black dashed lines indicate method



**Figure 3.12:** The expand/collapse mechanism using the example of the class *IvyConvertPom*.

inheritance relations between super and sub methods. The color codes of the methods are the same as for the JAVA MAP. The Type Hierarchy View provides the same basic navigation features as the rest of the map, including zoom and an outline view with a thumbnail, as well as a section showing the color codes of the methods. As we can see in Figure 3.14, the selected class has one abstract superclass which partially implements two interfaces. The one method not implemented by the superclass is implemented by our selected class as indicated by the dashed line from the defining method of the interface *ConflictManger* to the purple method of the selected class. Furthermore, our class overrides one method defined in the abstract superclass. The selected class just has one class extending it. The subclass has one method which extends the method of the selected class and two methods which override the ones of their superclass.

The Type Hierarchy View is an easy way to quickly analyze the inheritance structure of a given element. It can be used to estimate the impact of superclass changes to locate unused code or serve as an input for refactorings.



**Figure 3.13:** A shape header of a class which is part in an inheritance hierarchy.

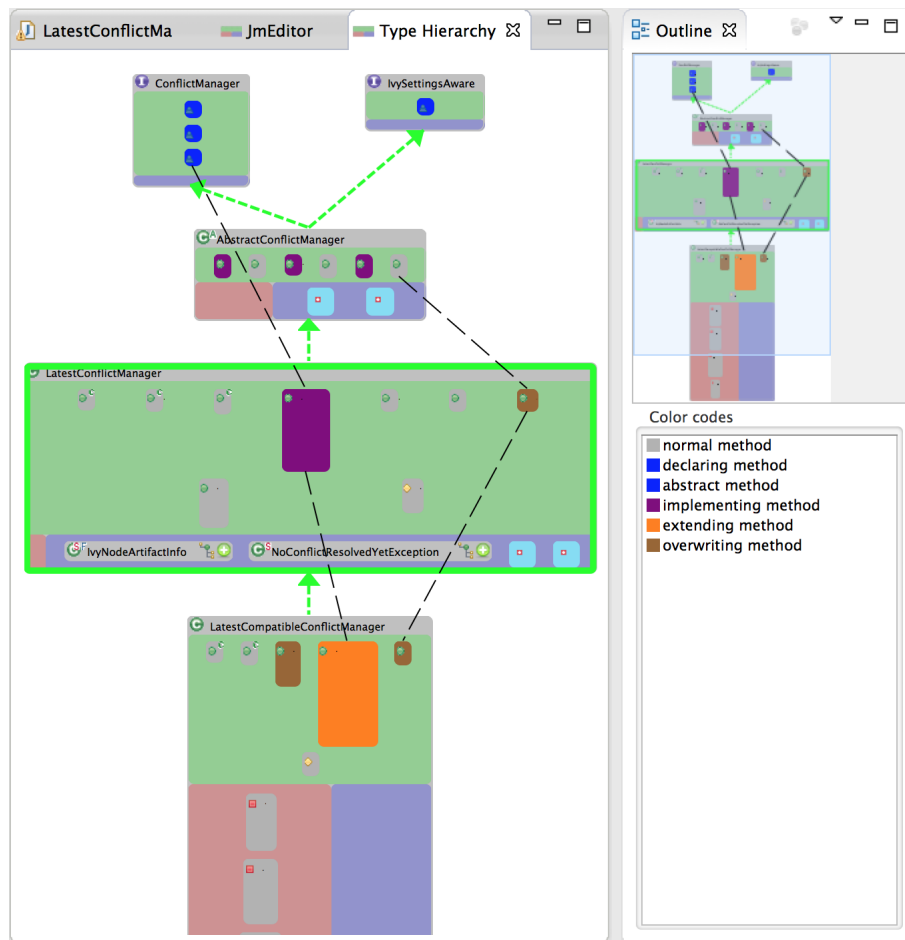


Figure 3.14: Example of a Type Hierarchy View.



## 3.4 Timeline View

In the Timeline View, all analyzed commits of a project are shown from past (left side) to present (right side), as depicted in Figure 3.15. Each commit is represented by a gray circle and label showing the short name of the commit. The graphs indicate the relative amount of changes. Green stands for the amount of added files, red for the amount of removed files, the blue graph indicates how many files have changed and the orange graph indicates how many entities (fields, methods, classes, etc.) have changed in the respective commit. The user can specify a time window by setting the left and right sliders to the according commits. The table below the Timeline contains all commits of the selected time window and provides additional information. The currently selected time window of the timeline serves as input for other features of the JAVA MAP like the Tree Rings View for example.

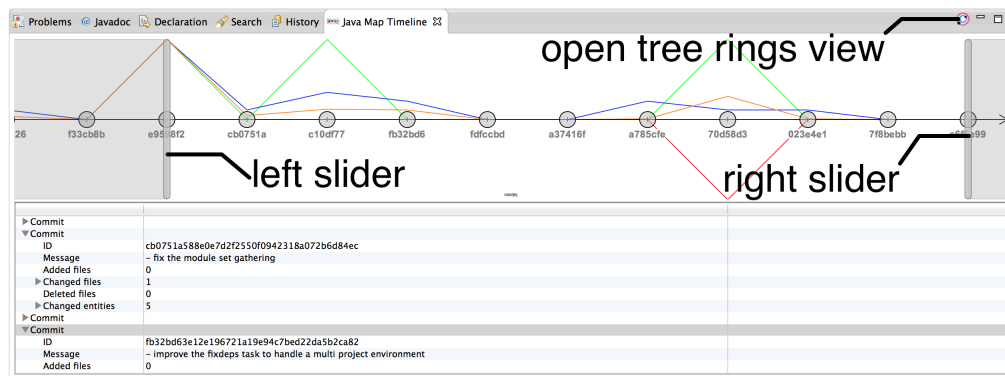


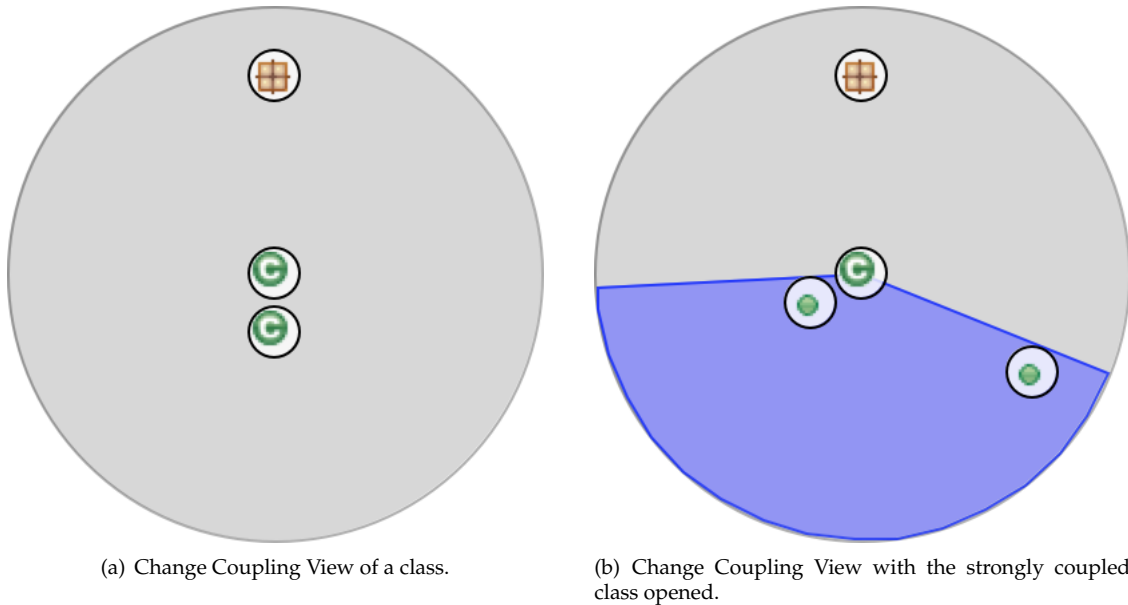
Figure 3.15: Example of the Timeline View of the JAVA MAP.

## 3.5 Change Coupling View

The Change Coupling View was inspired by the EVOLUTION RADAR described by D'Ambros et al. in [DLL09]. It allows the user to analyze how strongly different element of a system are coupled based on simultaneous changes made to both of them. For examples, if two methods of different classes are dependent on each other, e.g., through a code duplicate, changes to one need to be applied to the other as well. This leads to several commits, containing both methods. While code duplicates are an extreme example, logical couplings as described by Gall et al. [GHJ98], may lead to similar coupling patterns over several commits of a history. The more often two elements change together (in the same commit), the more coupled they are.

To open the Change Couplings View, the user simply selects a class or method in the Package Explorer and uses the command *Open Radar* from the context menu. The Change Coupling View visualizes the couplings of the selected element, as depicted in Figure 3.16(a). In the center we have the element of interest, in our example a class. All elements which have changed in the time frame (defined in the Timeline View) at least once are displayed in the view. The closer the element is to the center, the stronger the coupling is. To further investigate where the coupling originates from, the user can drill down into the coupled elements by double clicking on them.

Figure 3.16(b) shows the Change Couplings View after the coupled class was opened. The blue sector indicates the container, in our case the just opened class. We can see that the class contains two methods, of which one is highly coupled with our focus element, while the other one is not.



**Figure 3.16:** Example of the Change Coupling View.

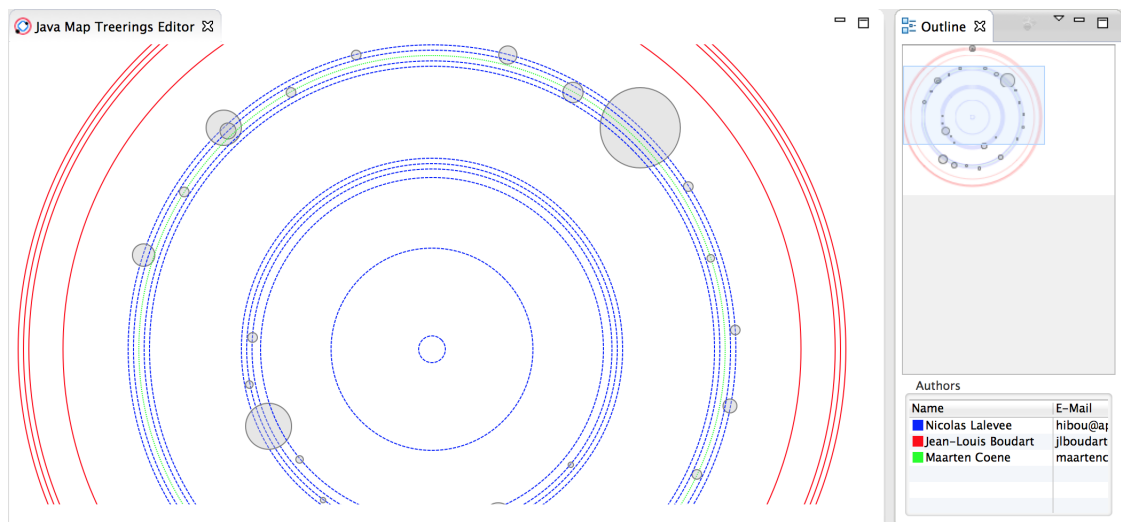
## 3.6 Tree Rings View

Based on the selected time window in the Timeline View, the user can generate a Tree Rings diagram by clicking on the *open tree rings view* button in the toolbar of the Timeline View. The JAVA MAP generates and displays an according diagram as depicted in Figure 3.17. Each ring represents one commit. They are ordered from newest in the center to the oldest at the rim of the diagram. The color and line style differ between the authors to make it easier to see who committed what, when. The distance between the circles indicate the relative amount of time passed between the commits: Close rings indicate commits which happened shortly after each others, while larger gaps indicate longer periods without any commits.

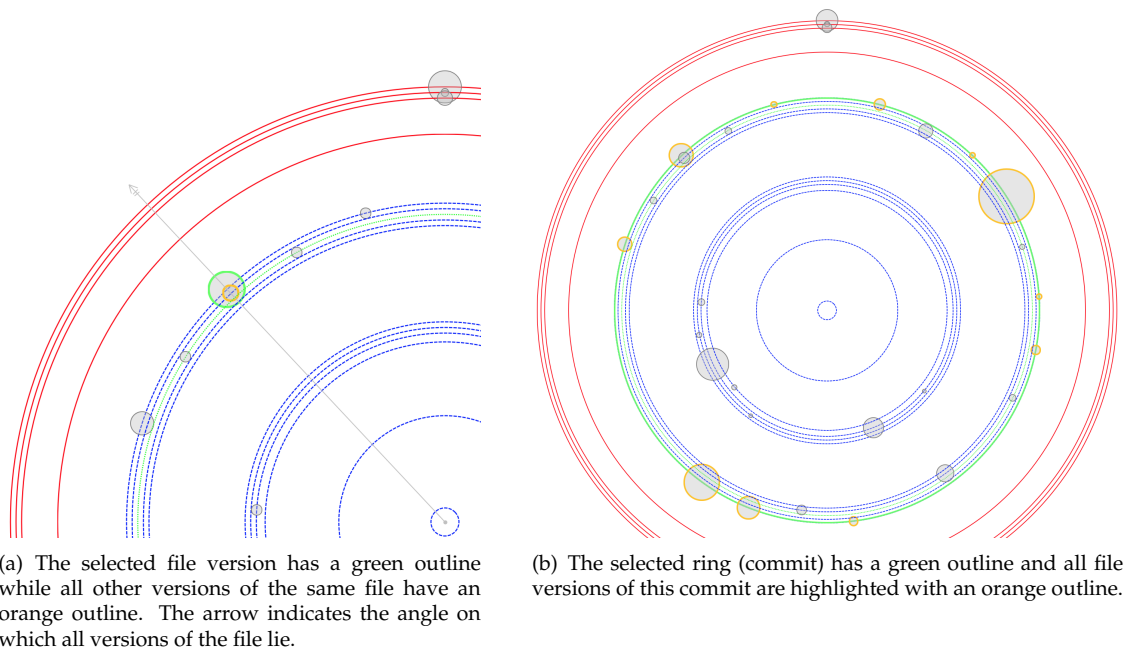
Each file which was changed at least once in the given time window gets a unique angle assigned. All versions of this file will be placed on an invisible axis (the axis will appear on selection) from the center to the rim with this angle. The gray circles on the rings represent one (changed) version of a file. The size of the circles depend on the number of changed entities (fields, methods, classes, etc.). Bigger circles indicate more changes to the same file e.g., several methods of a class were changed. The (version) circles are placed on their commit ring. Because all versions of a file are placed on their commit circle at the same angle. Several changes on different commits in the time window lead to a visual pattern much like a string of pearls. Selecting a file version causes the axes of the file to be displayed as shown in Figure 3.18(a). The selected version

is highlighted with a green border and all other versions of the file are highlighted with an orange border. On bigger and more complex diagrams, this helps the user to identify all changes of one file quickly. Selection is also supported for the commit rings as shown in Figure 3.18(b). The selected commit is highlighted with a green border and all file versions belonging to this commit are highlighted with an orange border. This helps the user to clearly see which files were changed in a certain commit, especially if a lot of commits were done in a short time frame which leads to a lot of close rings in the diagram.

Like all editors of the JAVA MAP, the Tree Rings View is equipped with a zoom functionality and an outline view as depicted in Figure 3.17. In the outline view, we have the standard thumbnail at the top, showing the whole diagram and a blue box indicating the current visible area of the view. On the bottom, a list containing the colors and names of all authors of the selected time window is shown. The Tree Rings View also supports hovering. For the commit rings, the short commit ID, the message header, the author and its email address are shown. For a file version, the file name, the commit and the names of all changed entities (e.g., methods, fields and inner types) are displayed.



**Figure 3.17:** Example of the Tree Rings View of the JAVA MAP.



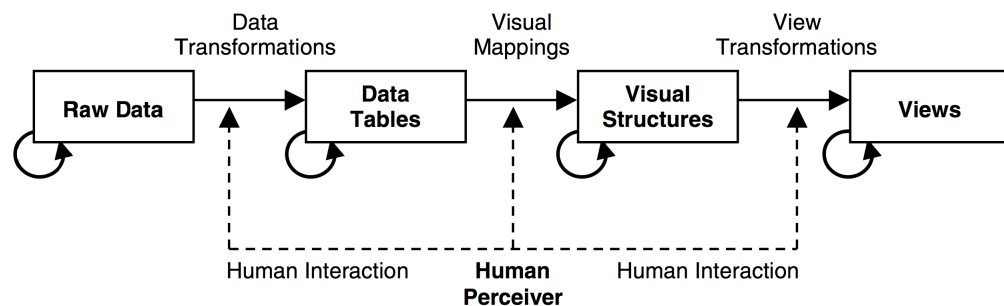
**Figure 3.18:** The selection mechanisms of the Tree Rings View.

# Implementation

This chapter focusses on the technical side of the JAVA MAP by briefly explaining its architecture and some of the implementation details. Section 4.1 introduces a reference model for software visualization. The basic architecture of the JAVA MAP are presented in Section 4.2, while Sections 4.2.1 to 4.2.4 explain the different components in more detail.

## 4.1 A Software Visualization Reference Model

The JAVA MAP follows the standard approach for software visualization by using a process involving several transformation and mapping steps to generate the representations presented to the user. A good reference model for visualization, as depicted in Figure 4.1, is described by Maletic et al. [MMC02]. It consists of two transformation and one mapping step, which together



**Raw Data:** idiosyncratic formats

**Data Tables:** relations (cases by variables) + meta data

**Visual Structures:** spatial substrates + marks + graphical properties

**Views:** graphical parameters (position, scaling, clipping, etc.)

**Figure 4.1:** A Reference Model for Software Visualization. (source: [MMC02])

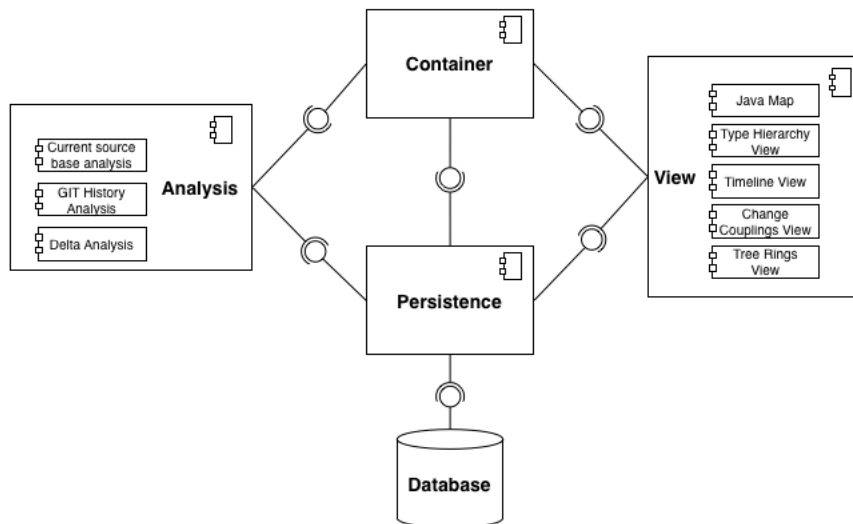
build the process chain of a visualization in the following way: *"The first transformation converts raw data into more usable data tables."* In the JAVA MAP, this transformation is done by the different analyzers. The raw data is the source code in the workspace and the history information pro-

vided by a GIT repository. *"Visual mappings transform the data tables into visual structures (graphical elements) [which are] the software specific visualizations we render [and are] typically very specific to a particular software engineering task."* [MMC02] The JAVA MAP uses the visual structures explained in Section 3.2, which are based on the concepts of the CLASS BLUEPRINT. As part of this mapping step, the persisted data from the analysis is processed and enriched by metadata. For example, the relative relationship between the siblings of a container is analyzed and stored in the form of positioning data. *"Finally, the view transformations create views of the visual structures by specifying parameters such as position, rotation, scaling, etc. User interaction controls the parameters of these transformations. The visualizations and their controls are all with respect to the application task."* [MMC02] In this final transformation step, the JAVA MAP uses a subset of the generated and persisted data to build the different views and editors described in Chapter 3.

The main difference between the JAVA MAP and the reference model depicted in Figure 4.1 is that the Map allows human (user) interaction by adjusting parameters only for the final transformation step.

## 4.2 Component Architecture of the Java Map

The overall component architecture of the JAVA MAP as depicted in Figure 4.2 is highly influenced by the individual steps of the process described in the previous section. The different components are built as individual ECLIPSE plug-ins, like the container and the persistence or as a bundle of plug-ins like the analysis and view parts. Each component will be briefly explained in the next sections.



**Figure 4.2:** The basic components of the JAVA MAP.

### 4.2.1 Container

The container (*ch.uzh.ifi.seal.javamap.container*) serves as a store for reused components like loggers<sup>1</sup>, layout managers<sup>2</sup> and basic utility libraries. Because there is no logic implemented in this plug-in, we named it *container* rather than *core* for example.

### 4.2.2 Analysis

As shown in Figure 4.2, the analysis consists of three different parts: The source base analysis of the current working directory, a delta analysis to keep the current model of the JAVA MAP up to date and a history analysis targeting GIT repositories. The first two analyzers are implemented in the plug-in *ch.uzh.ifi.seal.javamap.analysis*, the last one in the plug-in *ch.uzh.ifi.seal.javama.analysis.history*. All three analyzers use the same basic functionality (provided by the *ch.uzh.ifi.seal.javamap.analysis* plug-in) to analyze the source code objects and persisting the results in the database. The main classes are the *ch.uzh.ifi.seal.javamap.analysis.ast.ASTAnalyzer*, which extracts the detailed information from the abstract syntax tree (AST) created for the source entity and the class *ch.uzh.ifi.seal.javamap.analysis.Analyzer*, which controls the analysis process and stores the results in the database. The main difference between the three analyses is their controlling process and focus. The source base analysis simply runs through all files of the selected project and creates a model representing the source code entities in the data base. The delta analyzer registers itself as a listener on file changes in the Eclipse environment, runs the analysis only on the changed files and updates a given model in the database accordingly. The last and most complex analyzer builds a temporary Java project for each commit, based on the history information. All changed files are passed to the *ASTAnalyzer* and the fine grained differences are persisted in the database.

### 4.2.3 Persistence

The persistence of the JAVA MAP is implemented in the plug-in *ch.uzh.ifi.seal.javamap.persistence*. The model itself is designed using the ECLIPSE MODELING FRAMEWORK<sup>3</sup> (EMF) described by Steinberg et al. [SBPM08]. For the object-relational mapping (ORM) HIBERNATE<sup>4</sup> is used, in conjunction with TENEO<sup>5</sup> to automate the mapping process. The tool chain is completed by a set of Data Access Object (DAO) implementations, which provide a convenient access to the persistence elements and serve as the facade to persistence layer. An UML diagram of the complete persistence model is shown in Figure 4.4. This tool chain setup allowed us to quickly adapt our persistence model to changes if needed by simply adjusting the UML diagram and regenerate the complete persistence module. For more information and technical details we like to refer the reader to our previous work described in [Web13].

One of the strong points about our current persistence model is its versioning support for the persisted elements. The versioning is enabled by two elements. First, each entity has an id, which is simply the full qualified path from the project root to the element in question. Second, the element is attached to the commits it is part of. If an element changes from one commit the next, a new version of the element is created and added to the persistence model. With this technique only the changes need to be persisted but at the same time we can recreate the complete model for any given commit. An example is shown in Figure 4.3(a). The element *ClassA* with the id */Project/src/packageA/ClassA* is first encountered in *commit 11*. A corresponding version is created

<sup>1</sup>The JAVA MAP uses the apache Log4J logging framework ([logging.apache.org/log4j/](http://logging.apache.org/log4j/))

<sup>2</sup><http://www.miglayout.com>

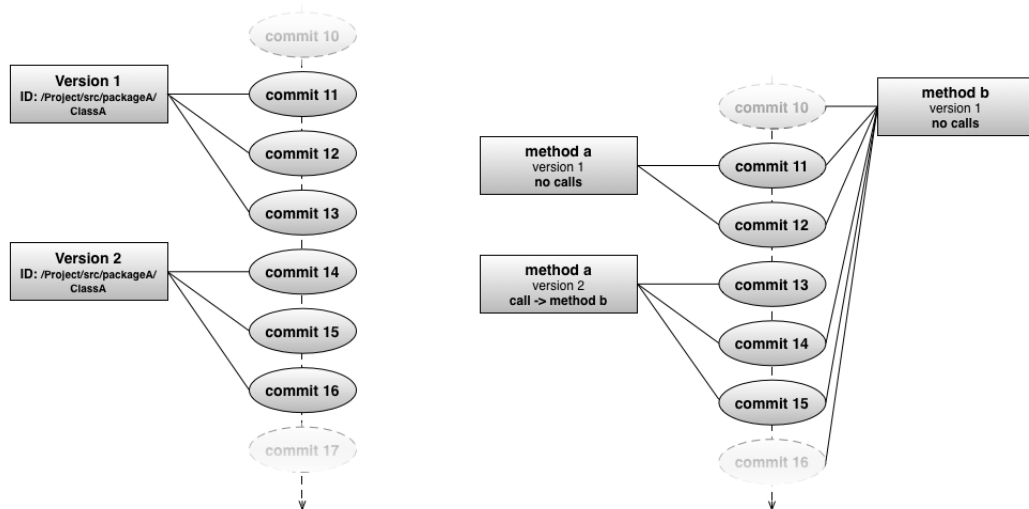
<sup>3</sup><http://www.eclipse.org/modeling/emf/>

<sup>4</sup><http://hibernate.org/>

<sup>5</sup><http://wiki.eclipse.org/Teneo/Hibernate>

in the database and *commit 11* is attached to it. For the next two commits (12 & 13) the element is not changed, so the two commits are attached to the same version (*version 1*) of the element. In *commit 14*, the element changed. This leads to the creation of a new *version 2* of the element and an attachment of *commit 14* to this new version. For the next two commits (15 & 16) we have again no change and simply attach the corresponding commits again to *version 2* of the element. Finally the element was deleted in *commit 17*. This leads to no new version, but also no attachment of the corresponding commit to any version of the element from this point on. The DAOs of the persistence facade allow direct access to any version of an element by specifying its id and the commit.

This mechanism allows us to decouple elements which are differently related in different versions. Let us have a look at the example of a method call as depicted in Figure 4.3(b). Before *commit 11*, the two methods are not related at all. In *commit 13* *method a* is changed and newly has a call to *method b*. Therefore, a new version of *method a* is created which, beside all other changes, has an additional call object targeting at *method b*. Although from *commit 13* on *method b* is called by *method a*, we do not need to create a new version of *method b*. The new version of *method a*, together with some logic implemented in the DAOs, allows us to represent this bidirectional relationship by only creating a new version of the 'physically' changed element. This decoupling mechanism works for all kinds of relations like inheritance, method calls, field accesses and even parent child relationships.



(a) The 'labeling' mechanism of the JAVA MAP persistence.

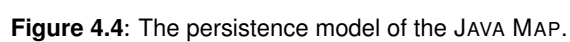
(b) Example for the decoupling of related methods.

**Figure 4.3:** Illustration of the versioning mechanism of the JAVA MAP persistence.

#### 4.2.4 View

The visual part of the JAVA MAP is implemented as several individual plug-ins. All visualizations use EMF to create the view model and the GRAPHICAL EDITING FRAMEWORK (GEF) as





described by Rubel et al. [RWC12] to create the controller and view elements. The information exchange between the different editors and views is done using the SELECTION FRAMEWORK of ECLIPSE and was inspired by the work of Hoffmann [Hof08]. The different visualizations are implemented in the following plug-ins:

The JAVA MAP and the Type Hierarchy View are implemented in *ch.uzh.ifi.seal.javamap.view.model* and *ch.uzh.ifi.seal.javamap.view*. Because the map is based on our previous work described in [Web11], the view model and the visualization are still separated in two different plug-ins. The Type Hierarchy View was added to the Map for two reasons: First, both visualizations use the same basic set of shapes (e.g., classifiers, methods, fields, etc.), and second, the Type Hierarchy View can reuse a great part of the controllers from the Map as well.

The Change Coupling View is implemented in the plug-in *ch.uzh.ifi.seal.javamap.radar*. The strange name stems from the initial idea to build an animated radar like visualization. Unfortunately, the prototype turned out to be pointless and simply confuses the user. Therefore, the Change Coupling View was changed into the prototype we have now and only the odd name remained.

The history visualizations of the Timeline View and the Tree Rings View are implemented in the plug-ins *ch.uzh.ifi.seal.javamap.timeline* and *ch.uzh.ifi.seal.javamap.treerings* respectively. Both consist of an own view model in EMF and a GEF editor for the visualization.

The underlying principle of all visualizations work on the same combination of EMF and a GEF editor embedded either in a ECLIPSE view or editor. Because we have already explained this visualization stack in depth in our previous work, we like to refer the reader to [Web11] for more information and implementation details.

# User Study

To evaluate if the JAVA MAP actually has a positive impact on the every day work of developers, we conducted a small user study at the end of our theses. In this Chapter we report our findings. Section 5.1 captures our motivation. The detailed set up of the study is introduced in Section 5.2. The different task are explained in Section 5.3. In Section 5.4 we present the results of the study and the chapter is closed with a short discussion of the user feedback in Section 5.5.

## 5.1 Motivation

The goal of this evaluation was twofold: First, we were interested, whether the JAVA MAP offers a real benefit to the user, in the form of reduced time needed to solve certain development task. So our research question was: *"Is there a significant difference in the completion time of a task between a developer using the JAVA MAP and a one just using the normal tools provided by the Eclipse IDE?"* Second, we were interested, in how people react to the JAVA MAP in general. This included questions like *"How fast are they able to understand the graphical representations and use the navigation features to solve the tasks?"*, *"How fast they adapt to the new tool?"*, *"Are they starting to think in the graphical domain presented?"*.

Due to time limitations, we only evaluate the first question statistically and simply report user reactions and comments for the later ones.

## 5.2 Study Setup

In this section we explain the setup and procedure of our study in detail. We designed the study to fit a small number of subjects. A *Within Subjects Design* was used, where each participant answers the same questions twice, once with and once without the JAVA MAP. The study was conducted with each participant individually and consisted of the following five parts:

- Explanation of the tasks (5 minutes)
- A quick introduction into the JAVA MAP (10-15 minutes)
- A test set to be solved with the JAVA MAP (maximal 20 minutes)
- A test set to be solved without the JAVA MAP (maximal 20 minutes)
- A quick roundup (5 minutes)

At the beginning of the evaluation, we introduced the participant to the tasks, by quickly explaining each question. This was done to ensure that the participant knows what he will be asked to do, while actually solving the tasks. The introduction to the JAVA MAP comprised of a short summary of the visual elements, navigation and different features and was given just before the test set with the JAVA MAP was solved (see table 5.1). Both test sets consisted of the exact same questions, but have different targets. For example one test set asks questions about 'ClassA', while the other set targets 'ClassB'. This means, each participant solved the same questions, once with and once without the map. We will refer to the two test sets as 'SetA' and 'SetB'. The detailed difference between the two sets are explained for each question respectively in Section 5.3. Each session was finished with a short open discussion about the evaluation and the JAVA MAP.

### Test Groups

Because each question has to be solved twice, there could be a learning curve which may lead to slightly faster solving times in the second run. For example questions may be read faster, as reading the first part of a question may be enough to remember what is asked. Or the participant could start to develop a solution path before the time measurement actually starts. To minimize the impact of such effects over all participants, we took two countermeasures. First, we explained the questions of a different test set to the participant (*Explanation of the tasks*), thereby giving him the chance to ask questions regarding the tasks. So the questions (but not the actual targets) were known before solving the first test set. Second, we created four different test groups A - D with their corresponding test procedures as shown in table 5.1. Thereby, the same test set was solved once first and once second, once with and once without the JAVA MAP. By mixing the test sets in the four procedures we tried to equate the impacts of possibly occurring learning effects as much as possible.

### Test Sets

Each test set consists of twelve tasks grouped into four blocks containing three questions each. The individual tasks together with their rationale are explained in Section 5.3. For each block a time limit of five minutes was assigned. The participant was allowed to quit a block at any time, which resulted in the maximum time (5 minutes) assigned to the block in total and an unsolved mark for all remaining questions of that block.

The task blocks to be solved with the JAVA MAP contained an additional first '*Preparation*' paragraph. These paragraphs explained how to open the needed views of the JAVA MAP for the task. Because no participant had prior experience with the JAVA MAP, this help was given. The rationale is that we don't want to measure how well the participants could follow the quick introduction of the JAVA MAP, but rather how well the JAVA MAP supports the user in solving the given task. The time measurement started once the steps of the preparation paragraph were completed.

### Technical Setup

Each participant solved the question on the same laptop. The baseline toolset used to solve the question without the JAVA MAP consisted of the Eclipse Classic package (v4.3.1) including the EGit plugin<sup>1</sup> and an EXCEL spreadsheet containing all metrics of the study object. To solve the question with the JAVA MAP a similar ECLIPSE package was used with an installed version of the JAVA MAP plugin.

### Research Population

As suggested by Wettel et al. [WLR11], we included users from both academia and industry. We had a total of 16 participants with the following backgrounds: Four of them are professional soft-

---

<sup>1</sup><https://www.eclipse.org/egit/>

Group	Test Procedure
A	<ol style="list-style-type: none"> <li>1. Explanation of the questions</li> <li>2. Solve SetA without the JAVA MAP</li> <li>3. Introduction into the JAVA MAP</li> <li>4. Solve SetB with the JAVA MAP</li> <li>5. Roundup</li> </ol>
B	<ol style="list-style-type: none"> <li>1. Explanation of the questions</li> <li>2. Introduction into the JAVA MAP</li> <li>3. Solve SetA with the JAVA MAP</li> <li>4. Solve SetB without the JAVA MAP</li> <li>5. Roundup</li> </ol>
C	<ol style="list-style-type: none"> <li>1. Explanation of the questions</li> <li>2. Solve SetB without the JAVA MAP</li> <li>3. Introduction into the JAVA MAP</li> <li>4. Solve SetA with the JAVA MAP</li> <li>5. Roundup</li> </ol>
D	<ol style="list-style-type: none"> <li>1. Explanation of the questions</li> <li>2. Introduction into the JAVA MAP</li> <li>3. Solve SetB with the JAVA MAP</li> <li>4. Solve SetA without the JAVA MAP</li> <li>5. Roundup</li> </ol>

**Table 5.1:** Test procedures for the different test groups.

ware developers with 4 - 10 years of experience. Six are master students of which three have worked part time in industry for at least two years. One is a Ph.D. student and five are postdoctoral researchers.

### Study Object

We used the Apache IVY project<sup>2</sup> to select the targets for the questions described in Section 5.3. At the time of our study, the current version of Ivy was 2.3, consisting of approximately 107,000 lines of code in 793 Java classes.

## 5.3 Tasks

Both test sets (SetA and SetB) contain the same twelve tasks, which are grouped into four blocks of three questions each. The next four subsections present each block, explain the questions, their rationale, intentions, as well as the different question targets.

### 5.3.1 History (Coworkers)

The questions in this block are centered around the recent project history.

#### Question Set Differences

<sup>2</sup><https://ant.apache.org/ivy/>

The targets of the two question sets differ solely in the time window: SetA asks the questions for the time window of the last 10 commits, while SetB targets the time window of the last 15 commits.

### Focus & Rationale

The rationale for these questions is based on the work of [FM10], [KDV07] and [dAM08]: Developers working in a collaborative environment, which most software projects are nowadays, need to know what the other team member did. Questions often asked by developers in this context are *"What have my coworkers been doing?"* [FM10], *"How have resources I depend on changed?"* [KDV07] or *"Who last changed this code?"* [dAM08]. Based on these questions, we defined our tasks shown in Table 5.2.

Question-ID	Question
H.1.1.A & H.1.1.B	How many authors have worked on the project in the last 10 (15) commits?
H.1.2.A & H.1.2.B	Which author contributed the most in this time frame?
H.1.2.A & H.1.2.B	Which file version has the largest number of changes (according to the number of changed methods and fields) in this time frame? Name the file and the commit.

**Table 5.2:** The questions of the history block.

The first question (H.1.1.A&B) simply asks who contributed something in the given time frame. This information can be used to identify the persons to ask or estimate the changed entities to expect, according to the working areas of the different authors. Therefore, the set of authors which recently committed changes to a project are a commonly used source of information. The second question (H.1.2.A&B) is a refinement of the first one by identifying the author which contributed the most. This can be seen as searching the source of the strongest impact in the given time frame. Although this question is rather vague (what precisely is meant by *"contribution?"*), all participants substituted 'contribution' by 'commit'. So the question actually answered by all participants was *"Which author made the most commits in this time frame?"* The third question (H.1.3.A&B) is different in two ways. First, it is rather specialized. Second, it is unfair! The question asks which was the biggest change done to one single file over the defined time frame, which is simply the change hotspot in the given time window. This information is useful in a lot of different cases, like getting an overview of the recent changes or identifying the main contributors for a certain area. However this question is unsolvable in the given time without a tool! We decided to ask this question here for two reasons. First, we wanted to show that a software visualization tool like the JAVA MAP not just supports you at the usual task, but also offers a lot of new and more refined information. Second, we wanted to check the reactions of the users.

## 5.3.2 Type Hierarchy

This block is centered around the understanding of type hierarchies and more precise around method inheritance.

### Question Set Differences

The two questions sets differ solely in the class to analyze: SetA asks questions about the class

*org.apache.ivy.plugins.version.ChainVersionMatcher*, while SetB targets the class *org.apache.ivy.plugins.version.LatestVersionMatcher*.

### Focus & Rationale

The tasks are derived from the question “Who implements this interface or these abstract methods?” defined by Sillito et al. [SMV08]. Type hierarchies are powerful instruments in object oriented code, allowing polymorphism and code reuse. Because inheritance relations build a central concept in object oriented code, understanding these relations are crucial to understand the internal mechanisms of a system. It allows powerful design constructs and is therefore at the center of many object oriented design pattern as described by [GHJV95]. However, the concepts of inheritance hierarchies bring an additional level of complexity to a software system. In order to successfully change and extend a given inheritance structure, it is important to understand its building blocks and their relationship among each other. The questions defined by Sillito et al. focus on several different aspects of type hierarchies, like finding all implementers of an interface, all subclasses or siblings of a class, as well as field access and more. For this evaluation we restricted the focus to one class and the concept method inheritance by asking the question shown in Table 5.3. The rationale for this decision is that the full understanding of a class subclass relationship, including the inheritance relation of their methods is the first step in the process of understanding a complete class hierarchy. At the same time, asking only about the properties of members of one class simplifies the task and makes it faster solvable.

Question-ID	Question
T.2.1.A & T.2.1.B	Name all methods extending the behavior of their super method.
T.2.2.A & T.2.2.B	Name all methods overriding the behavior of their super method.
T.2.3.A & T.2.3.B	Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

**Table 5.3:** The questions of the type hierarchy block.

The first question (T.2.1.A&B) asks for all methods which extend the behavior defined by their super method. This is a method with the same signature like its super method leading to an overriding but additionally containing a super call in its body. This technically leads to an extension of the behavior defined in the super method. The second question (T.2.2.A&B) in contrast asks for all methods overriding the behavior of their super method. The small difference of the super call distinguishing between extending or overriding may have a huge impact. If for example the super method is changing the values of fields of the class, this leads to a state change of the object. Extending means the child object will do the state change as well, while overriding does not, resulting in a complete different behavior. The third question (T.2.3.A&B) asks for all methods which are defined in an implemented interface or abstract superclass.

### 5.3.3 Entity Access

This block focuses on entity accesses and call dependencies.

#### Question Set Differences

The two question sets differ in the class, as well as in the field of the class to analyze: SetA asks questions about the class *org.apache.ivy.tools.analyser.JarModule* and the field 'mrid', while SetB targets the class *org.apache.ivy.plugins.version.Match* and its field 'revision'.

### Focus & Rationale

The questions are derived from the work of [FM10], [SMV08] and [dAM08]. In the process of code analysis, developers are interested in the dependencies between objects and therefore ask questions like "Where is this method called or type referenced?" or "What fields does this type or method access?". Entity accesses build up call hierarchies and thereby dependency networks between the elements of the system. On a larger scale they define how the components of a system interact with each other. Understanding these access and call dependencies is an important part in the overall system understanding. The questions of our evaluation focused on both field access as well as method calls, as shown in table 5.4. The first question (E.3.1.A&B) is derived directly from the question "Where is the value of this field retrieved?" [dAM08]. The same holds for the second question which is a small extension of the question "Where is the value of this field set?" [dAM08]. In addition to the direct access we also ask for indirect access (over getters and setters) if available. The rationale behind this is that most fields are declared private, which follows the encapsulation paradigm. In order to evaluate which elements of a system are using or changing the value of a field, we therefore also have to take this indirect accesses into account.

Question-ID	Question
E.3.1.A	For the field "mrid" of the class <i>JarModule</i> ( <i>org.apache.ivy.tools.analyser.JarModule</i> ), name all methods, which have a direct or indirect (over getter) read access.
E.3.1.B	For the field "revision" of the class <i>Match</i> ( <i>org.apache.ivy.plugins.version.Match</i> ), name all methods, which have a direct or indirect (over getter) read access.
E.3.2.A	For the field "mrid" of the class <i>JarModule</i> ( <i>org.apache.ivy.tools.analyser.JarModule</i> ), name all methods, which have a direct or indirect (over setter) write access.
E.3.2.B	For the field "revision" of the class <i>Match</i> ( <i>org.apache.ivy.plugins.version.Match</i> ), name all methods, which have a direct or indirect (over setter) write access.
E.3.3.A	Which methods of the class <i>JarModule</i> ( <i>org.apache.ivy.tools.analyser.JarModule</i> ) are called by test classes (classes contained in the package <i>test/java</i> )?
E.3.3.B	Which methods of the class <i>Match</i> ( <i>org.apache.ivy.plugins.version.Match</i> ) are called by test classes (classes contained in the package <i>test/java</i> )?

**Table 5.4:** The questions of the entity access block.



### 5.3.4 Metrics

#### Question Set Differences

Both sets ask for the same metrics but on different targets. The questions are shown in Table 5.5. The main difference, however, was the tool to be used. Because the basic Eclipse IDE does not provide any software metrics, a separate EXCEL sheet was provided to answer the question without the map.

#### Focus & Rationale

The last block of questions is centered around pure numbers. Software metrics capture countable attributes of source code entities. Examples are the lines of code (LOC) an entity consists of, or the number of calls to (FANIN) or from (FANOUT) an element, how many children an element contains and more. In contrast to the previous three blocks, the questions in this block are not directly derived from questions often asked by developers as described by Sillito, Murphy, Alwis et al. The main reason is that one metric alone is of little to no use, so they need to be collected, aggregated and interpreted. Therefore, metrics are normally found in reports and analysis, which is not where they emerge from and definitively not the main working area of developers. But as described by Lanza and Marinescu [LM06], metrics could be quite useful even on a low and fine grained level if they are presented in the right context. For example, knowing that a class consists of 500 lines of code won't let us assume much. But if we additionally know that the class only contains two methods, it becomes striking that the functionality in this class is not well structured. On the other hand, if the same class contains 50 methods the picture looks different: Now we know that the functionality is highly modularized, but at the same time, we may get the impression that this class is overloaded in terms of the different functionality it provides. So overall metrics can help us in getting a more precise overview of an element in question. The questions of the two sets are shown in Table 5.5

Beside this, there was a second rationale for this block: Because the JAVA MAP presents the metrics of the current selection, the user needs to navigate to the element in question in order to solve the task. This means the faster one can navigate in the map, the shorter the solving time. So the second interesting question for this special block was: After just 10 minutes of introduction and about 10 more minutes of solving the prior tasks, have the users gained enough experience with the JAVA MAP to solve the task as quickly or even faster compared to solving the task with a simple spreadsheet?

## 5.4 Evaluation Results

In this section we present the result of our empirical study. We start with a basic overview by showing the statistical analysis of the aggregated completion times of each participant in Section 5.4.1. We then go one step deeper and present the individual results obtained for each task in Section 5.4.2, together with our analysis and interpretation to explain how each task contributed to the overall result. As mentioned in Section 5.3.1 the last question of the history block is unsolvable without the JAVA MAP. Because the time measurements of the according two questions H.1.3.A and H.1.3.B would distort the overall result in an unfair way, we exclude them from our statistical analysis.

We use the non-parametric *Independent Samples Mann-Whitney U* (MWU) test with a significance level  $\alpha = 0.05$  as a statistical test for our hypothesis shown in table 5.6. The reason for this choice is that the *Shapiro-Wilk Test of Normality*, as well as the visual analysis of the histograms and the Q-Q plots, raised reasonable doubts about the normal distribution of our sample sets, especially when looking at the individual distributions for each task. Because the MWU test is nearly as

Question-ID	Question
M.4.1.A	Name the total number of classes (NOC) in the package <i>org.apache.ivy.core</i> .
M.4.1.B	Name the total number of classes (NOC) in the package <i>org.apache.ivy.plugins</i> .
M.4.2.A	Name the total number of lines of code (LOC) in the class <i>VersionRangeMatcher</i> ( <i>org.apache.ivy.plugins.version.VersionRangeMatcher</i> ).
M.4.2.B	Name the total number of lines of code (LOC) in the class <i>ResolveReportTest</i> ( <i>org.apache.ivy.core.report.ResolveReportTest</i> ).
M.4.3.A	Name the number of calls(FAN_OUT) made by the elements of the package <i>org.apache.ivy.plugins.trigger</i> .
M.4.3.B	Name the number of calls (FAN_OUT) made by the elements of the package <i>org.apache.ivy.plugins.report</i> .

**Table 5.5:** The questions of the metrics block.

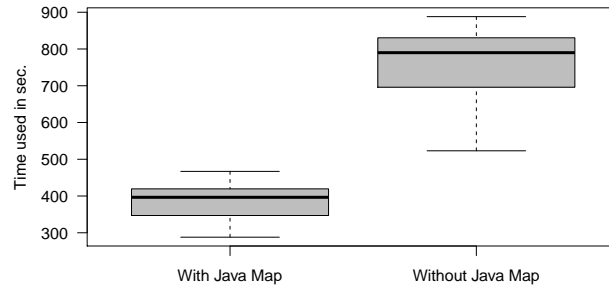
Hypothese	Alternative Hypothese
$H_0$ The distribution of the average number of seconds spent for solving a task is the same for the group using the JAVA MAP and the group not using the JAVA MAP.	$H_A$ The distribution of the average number of seconds spent for solving a task is different between the group using the JAVA MAP and the group not using the JAVA MAP.

**Table 5.6:** Test Hypotheses.

efficient as the t-test on normal distributions, but has greater efficiency than the t-test on non-normal distributions, we decided to use the MWU test to be on the save side.

## 5.4.1 Overview

In this section we present the statistical analysis of the aggregated completion times. For each participant we summed up the time needed to solve each question set. This leads to 16 time measurements with, and 16 without the map. The question sets were solved on average 50% faster with the JAVA MAP than without. The mean total solving time with the JAVA MAP was 6 min 23 sec with a standard deviation of 53 sec and a median of 6 min 36 sec. For the total solving times without the JAVA MAP we observed a mean of 12 min 41 sec with a standard deviation of 1 min 39 sec. and a median of 13 min 10 sec. This is equivalent to an average time saving of 6 minutes and 18 seconds for all 11 tasks. Figure 5.1 further illustrates that the complete box plot with the JAVA MAP is below the one without the JAVA MAP. In other words, all participants were faster in solving the given tasks when using the JAVA MAP, than anyone not using the JAVA MAP.



**Figure 5.1:** Box plot of the cumulated completion times.

### 5.4.2 Detailed Task Analysis and Interpretation

The results presented in Section 5.4.1 show a clear advantage for the users with the JAVA MAP compared to the standard functionalities provided by the ECLIPSE IDE. In this section, we break down the analysis to each individual task, analyze their influence on the overall outcome and present our interpretation of the observations. For all detailed analysis the non-parametric *Independent Samples Mann-Whitney U* (MWU) test was used with a confidence level of 95% and the *Bonferroni-Holm* method was applied to the results to counteract the problem of multiple comparisons. A summary of our observations is given in table 5.7.

#### History (H.1.1)

This task focusses on historical information of the project in a certain time frame (the last 10 or 15 commits). The question is, how many authors have worked on the project in this time frame by committing at least one change. Figure 5.2 shows a box plot of the completion times with and without the JAVA MAP.

#### Results

The 75<sup>th</sup> percentile of the box plot for the JAVA MAP group is below the box plot of the control group. In other words, three quarters of all subjects with the JAVA MAP were faster in solving the task than anyone using the baseline tool. The Mann-Whitney U test shows that the participants were able to solve the task significantly faster when using the JAVA MAP. With a corrected  $\alpha=0.05$ , we have a p-value of 0.001083526. With the tool, the users were able to solve the task in 26 seconds on average, in contrast to 59 seconds without the JAVA MAP.

#### Interpretation

We registered that most participants double checked their answer when not using the JAVA MAP. The standard history view of Eclipse presents the information in a list. Several users seemed unsure about the correctness of their answer and therefore scrolled forth and back through the list to check if they had missed an author or not. While using the JAVA MAP, participants initially took more time to just analyze the graphics, but once they started writing down the answer there was

Task ID	Using the Java Map	Min (Min:Sec)	Max (Min:Sec)	Mean (Min:Sec)	Median (Min:Sec)	StdDev (Min:Sec)	p-value
H.1.1	yes	00:09	01:06	00:26	00:22	00:16	0.001083526
	no	00:35	02:12	00:59	00:53	00:24	
H.1.2	yes	00:04	00:22	00:12	00:11	00:05	0.139851002
	no	00:08	00:45	00:19	00:16	00:11	
T.2.1	yes	00:14	00:43	00:22	00:20	00:09	1.67E-05
	no	00:51	03:10	01:54	01:58	00:42	
T.2.2	yes	00:16	00:45	00:32	00:35	00:09	0.001620717
	no	00:20	02:25	01:13	01:01	00:35	
T.2.3	yes	00:06	00:44	00:26	00:26	00:09	0.00160931
	no	00:11	02:59	01:11	01:00	00:43	
E.3.1	yes	00:31	01:24	00:58	00:58	00:15	0.066441103
	no	00:48	01:46	01:15	01:16	00:19	
E.3.2	yes	00:14	01:21	00:47	00:40	00:25	0.637530297
	no	00:19	01:10	00:41	00:37	00:14	
E.3.3	yes	00:26	01:06	00:45	00:47	00:11	0.000877571
	no	00:08	03:26	01:46	01:37	00:58	
M.4.1	yes	00:07	01:18	00:39	00:38	00:21	0.02810696
	no	00:37	01:58	01:09	01:09	00:28	
M.4.2	yes	00:21	01:45	00:39	00:34	00:21	0.066441103
	no	00:20	01:24	00:52	00:53	00:18	
M.4.3	yes	00:13	01:13	00:37	00:39	00:19	0.03128975
	no	00:25	02:38	01:05	00:54	00:35	

**Table 5.7:** Summary of the individual results per task.

no double-checking. We assume that the color pattern used to distinguish the different authors helps the user in this task. It seems to be easier to identify the differently colored rings in the view than the different names in the table. While the time frames were quite short, we already have a significant difference in the completion times. This result however does not give us any hints on how the tree rings view would perform on really large time frames (several hundreds of commits, with dozens of authors). Clearly the task would become more difficult with the baseline, but a lot of authors would also lead to many different color rings. Too many colors could confuse the user, rather than help him. However, for small time frames, the tree rings view has proven to be useful.

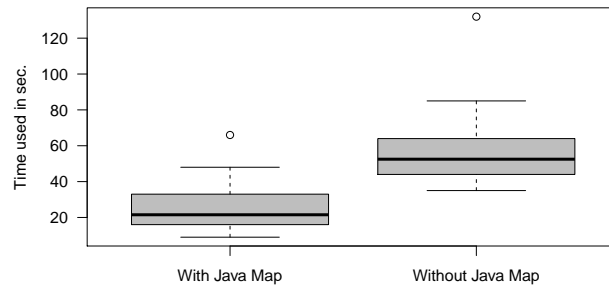
## History (H.1.2)

This task asked to name the author which contributed the most in the given time frame (last 10 or 15 commits). Although the question stated not clearly what ‘contributed’ means, all participants interpreted the question as “Name the author with the most commits in the time frame.” Figure 5.3 shows a box plot of the completion times with and without the JAVA MAP.

### Result

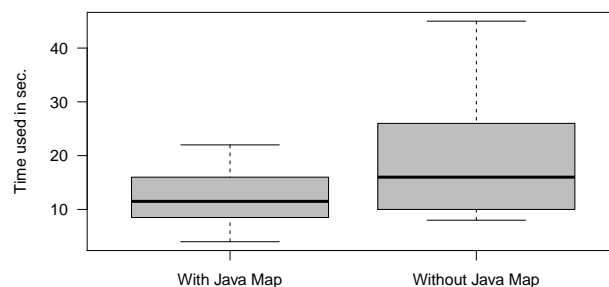
The difference in the completion times of the two groups is statistically insignificant. The task was solved by the JAVA MAP group in 12 seconds on average, while the control group had a mean of 19 seconds, with a standard deviation of 5 seconds and 11 seconds receptively.

### Interpretation



**Figure 5.2:** Box plot for Task H.1.1.

In both time frames there was one author which had about 80%-90% of all commits, which made it easy to identify him. Therefore, the task became extremely simple and both groups were almost equally fast in solving it. But while the JAVA MAP users just wrote down the name corresponding to the predominant color of the tree rings view, some of the control group users double checked their answers again. While this question was too simple and does not reveal any benefit in using the tool, if we imagine a large time frame again, it could be more likely one is able to name the predominant color faster than finding the name most stated in a table.



**Figure 5.3:** Box plot for Task H.1.2.

### Type Hierarchy (T.2.1)

This task asked to name all methods of a class, extending the behavior of their super method. These are all methods which override their super methods but contain a super call in their body. Figure 5.4 shows a box plot of the completion times with and without the JAVA MAP.

#### Result

The group using the JAVA MAP was significantly faster in solving the task than the control group (MWU test, corrected  $\alpha=0.05$ , p-value < 1.67E-05). The box plot shows that all users of the JAVA MAP (including the outliers), were faster than any user of the control group.

#### Interpretation

Understanding the inheritance relationship between a method and its super method is a (partially) manual task in the standard Eclipse IDE. Therefore, the users without the JAVA MAP had to navigate forth and back and read through the code. On the other hand, the JAVA MAP provides this information as part of its method representation by using a color scheme. This simplified the task for the users of the Map to just identify the methods with a certain color.

Because all participants solved the task with and without the JAVA MAP, we could ask them afterwards what they thought about the question. We asked them if they think the question is too onesided and favours our tool. All pointed out the clear advantage of the tool, but judged the question as fair. They all agreed that the understanding of type hierarchies is a central part of program comprehension and therefore this question is valid.

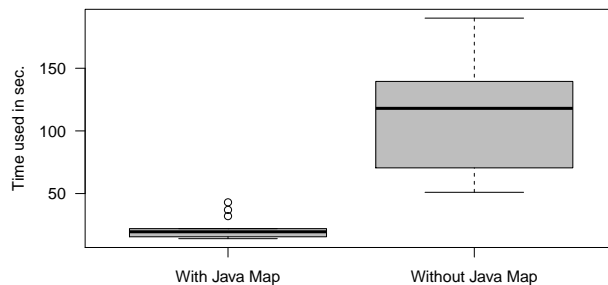


Figure 5.4: Box plot for Task T.2.1.

### Type Hierarchy (T.2.2)

This task asked to name all methods of a class, overriding the behavior of their super method. Figure 5.5 shows a box plot of the completion times with and without the JAVA MAP.

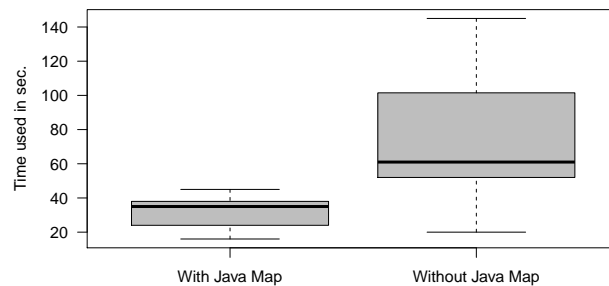
#### Result

This task was solved significantly faster by the group using the JAVA MAP. Using the Mann-Whitney U test with a corrected  $\alpha=0.05$  we get a p-value < 0.001620717. The box plot shows that

all users of the JAVA MAP were faster than three quarters of the control group.

### Interpretation

We still have a clear difference between the two groups, but it is not as striking as in the first task of this section. The main reason was that the control group was able to '*reuse*' some of the investigations they made for the previous task: Once the overriding methods are identified, all non-extending (no super call) fall into the answer set of this task. However, a mean of 1min 13sec for this task shows that the solution was not as trivial for most participants. We registered again a lot of double-checking. When asked about this question afterwards, many participants felt a heavy time pressure while doing the task without the JAVA MAP. They stated that they would need more time in a real situation to be absolutely sure to solve the task correctly.



**Figure 5.5:** Box plot for Task T.2.2.

### Type Hierarchy (T.2.3)

This task asked to name all methods of a class, which implement a definition of an inherited interface or an abstract definition of an abstract superclass. Figure 5.6 shows a box plot of the completion times for this task, with and without the JAVA MAP.

### Result

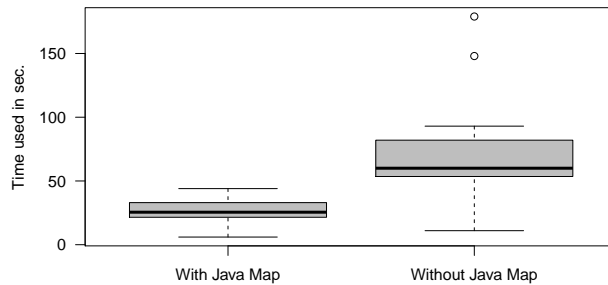
The group using the JAVA MAP was significantly faster in solving this task than the control group. The Mann-Whitney U test with a corrected  $\alpha=0.05$  gives us a p-value  $< 0.00160931$ . The box plot shows that all users of the JAVA MAP were faster than three quarters of the control group.

### Interpretation

Basically the interpretation and observation of the previous task holds for this one as well. Beside the two outliers the control group was faster in solving this task, because they could use the information collected by solving the previous two tasks. The two outliers in fact became unsure about the correctness of their previous answers when solving this task and double-checked all methods once more which cost them a lot of time.

For the control group, the professional software engineers were faster in solving these tree tasks,

as they have to do similar analysis in their daily work.



**Figure 5.6:** Box plot for Task T.2.3.

### Entity Access (E.3.1)

This task asked to name all methods having a direct or indirect (over a getter) read access. Figure 5.7 shows a box plot of the completion times for this task.

#### Result

The difference in the completion time between the two groups is statistically insignificant.

#### Interpretation

Finding accessors of a field is a common task and well known by all developer. The users of the control group had no problem in solving this task quickly. Although the group using the JAVA MAP was not slower than the control group, most users felt uncomfortable using the tool for this task. Two participants actually asked if they had to use the JAVA MAP. The users explained that they know the commands and tools in ECLIPSE to find references on elements and therefore do not see the need or benefit of the JAVA MAP.

### Entity Access (E.3.2)

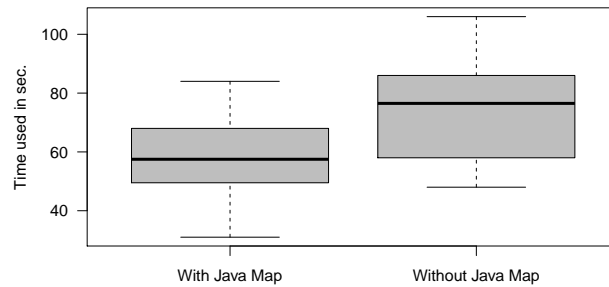
This task asked to name all methods having a direct or indirect (over a getter) write access. Figure 5.8 shows a box plot of the completion times for this task.

#### Result

The difference in the completion time between the two groups is statistically insignificant. The medians of the two groups are rather close (47 seconds with and 41 seconds without the JAVA MAP). But as we can see in the box plot, the standard deviation for the JAVA MAP group is much higher (25 seconds) compared to the control group (14 seconds).

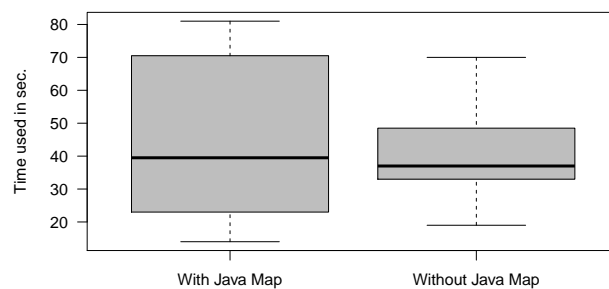
#### Interpretation





**Figure 5.7:** Box plot for Task E.3.1.

We observed the same situation as in the previous question. The higher standard deviation for the JAVA MAP group can be explained by the fact that one task target (*org.apache.ivy.plugins.version.Match*) has write accesses from a test classes. The test components are far away from the target class, which forces the user to navigate in the JAVA MAP by zooming, adjusting and opening elements. Some participants were quickly familiarized with the JAVA MAP and had no problems in solving the task, while others get lost in the Map and had to go back to the start and redo the navigation steps.



**Figure 5.8:** Box plot for Task E.3.2

### Entity Access (E.3.3)

This task asked for all members of the class which are directly accessed by test elements. Figure 5.9 shows a box plot of the completion times for this task.

#### Result

The group using the JAVA MAP was significantly faster in solving this task than the control group (MWU test, corrected  $\alpha=0.05$ , p-value < 0.000877571). The box plot shows that the 75<sup>th</sup> percentile of the JAVA MAP group is below the 25<sup>th</sup> percentile of the control group.

#### Interpretation

Instead of asking for all accesses on a single element, we asked for all accesses from a specific subset (test classes) to any elements of a class. While the question is just a small twist to the previous two, we have a completely different outcome. To answer this task with the tool, the same steps are needed as for the previous two tasks. Without the JAVA MAP, the user needs to check every method and field separately. Beside the time needed to manually check each reference, the users seemed to be less confident. Some again double-checked their answers to ensure they made no mistake.

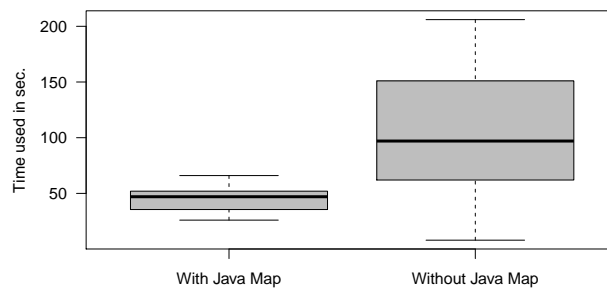


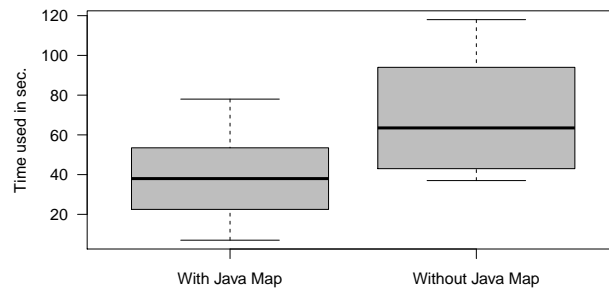
Figure 5.9: Box plot for Task E.3.3.

### Metrics (M.4.1)

This task asked for the number of classes (NOC) contained in a specific package. Figure 5.10 shows a box plot of the completion times. Because the Eclipse IDE does not provide metrics information, a spreadsheet containing all metrics of the study object (IVY project) was given to the control group to solve this task.

#### Result

The statistical test showed a significant difference in the completion times between the two groups (MWU test, corrected  $\alpha=0.05$ , p-value < 0.02810696). The box plot shows that three quarters of the JAVA MAP users solved the task faster than half of the control group.



**Figure 5.10:** Box plot for Task M.4.1.

### Interpretation

All tasks from the metrics block test two combined things: First, if the user is faster getting the metric value from the JAVA MAP, compared to searching for it in the spreadsheet. Second, how well the participant understands the navigation concepts of the map. In order to find the asked information, the user needs to navigate in the map or between the source code editor and the map. It is quite surprising to get a significantly better result for the JAVA MAP group already for the first task of this block. Although the task description for the JAVA MAP users included a small hint in how to solve the first question, most participants did not read it but just started solving the task. Some of them just navigated through the map and searched for the package in question, while others opened a class file of the package and moved from there out into the map.

### Metrics (M.4.2)

This task asked for the total lines of code (LOC) contained in a specific class. Figure 5.11 shows a box plot of the completion times of the two groups. The control group again used the provided spreadsheet to solve the task.

### Result

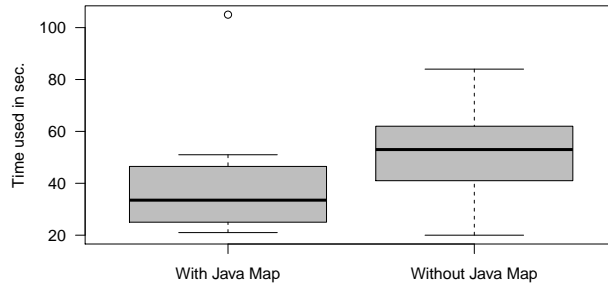
The statistical test showed no significant difference in the completion times between the two groups.

### Interpretation

What we have expected for task M.4.1 actually happened here. One person got stuck in the navigation process and needed much more time to solve the task than the rest of the group. We can see in the box plot, there is one outlier in the JAVA MAP group. Because of our small sample set, the impact is strong enough to force us to reject the null hypothesis for this task.

### Metrics (M.4.3)

This task asked for the the total number of calls (FAN\_OUT) made by the elements of one package. Figure 5.12 shows a box plot of the completion times.



**Figure 5.11:** Box plot for Task M.4.2.

### Result

The statistical test shows a significant difference in the completion times between the two groups (MWU test, corrected  $\alpha=0.05$ , p-value < 0.03128975)

### Interpretation

For this task we have the opposite situation compared to the previous task. As we can see in the box plot, the control group has two outliers. Again, because of the our small sample set of just 18 participants, the impact is strong enough, this time in the favor of the JAVA MAP.

All together, the three questions of the last task block show the smallest differences between the two groups. One reason may be the setup for the control group: The participants did not have to switch or start a different tool. They were given the spreadsheet before the time measurement started, so the three tasks boiled down to the interpretation of the table and finding the right rows in it. As we will show in the next section, the user feedback to these questions was actually much more diverse, than the statistical results would suspect.

## 5.5 User Feedback

As explained in Section 5.2 we held a short, informal discussion with each participant at the end of the evaluation session. We asked the same three questions to each participant and noted down their responses. The questions were: *"What do you like about the JAVA MAP? What are the weak points of the JAVA MAP? Where do you see potential for enhancement? What is missing?"* While the first two questions were mainly to evaluate the strength and weaknesses of the Map, the third question was targeting our second aim described in Section 5.1: Are the participants starting to think in the graphical domain presented? If so, the responses should be more vivid and target graphical and navigational issues as well. In the next three sections we present the paraphrased answers, given by the participants to these three questions.

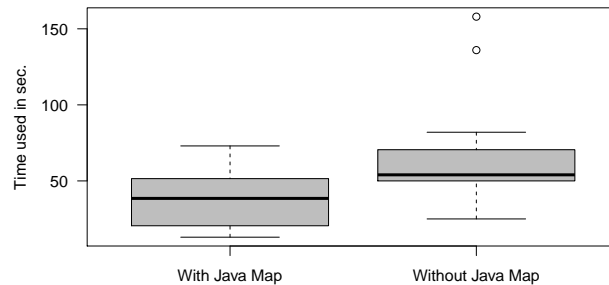


Figure 5.12: Box plot for Task M.4.3.

## Positive Feedback

The tight integration of the JAVA MAP into the ECLIPSE IDE was appreciated by all users. The animations of the map (zoom out to the map and the expand/collapse mechanism of the shapes) were specially mentioned by five of them in the discussion. They said that it helps them to keep the focus. The feature which allows to jump back into the code was rated as one of the strongest points, because it allows the user to investigate an element in detail if needed. The JAVA MAP was considered intuitive and easy to learn, in respect of coloring as well as shape sizing and positioning. Six participants specially highlighted the Type Hierarchy View, because in their eyes, something similar is missing in ECLIPSE. While most participants were skeptical about metrics beforehand, they liked the integration into the outline. We noted down quotes like *"if they are accessible where they belong to, they are actually quite useful!"* We registered several cases, where the metrics were used to get a precise understanding of an aspect by answering questions like *"how big is this actually?"* or *"how many calls are this?"*. Two users discovered the multiple selection capability of the Map by themselves and rated it as a *'big plus'*, because it allows to build a context of interest by highlighting the relations of several elements at the same time. The Timeline View and the Tree Rings View was highly appreciated by those participants with professional experience from industry. Some of them even asked if this part of the Map is available as a standalone tool. All participants saw the highest potential of the JAVA MAP in its power to support the user in getting a basic understanding and a quick overview of a system. They think it is perfectly suited for new team members and people which work in a project on an irregular basis. All but two participants showed great interest in the JAVA MAP and asked when and where it will be publicly available.

## Critical Points

Clearly the most criticized point was the performance. The time until hovers pop up and the reaction time for the selection of an element with many (several hundred) relations was considered to be slow. While no participant actually consider it as a *'show stopper'*, they clearly see room for improvement on this point. For one participant the Tree Rings View was not intuitive and several students mentioned that they do not see a great use for the Timeline and Tree Rings View. Most users tried to pan the view in the Map and were confused and disappointed about the lack of this

feature. Finally some participants criticized the huge size of the map, but reminding them about the zoom function mainly solved this issue right away.

### **Suggestions for further enhancements**

When asked what is missing or should be enhanced next, most participants started by themselves to play with the JAVA MAP again. Most of them asked for a panning feature, some for other navigation helps like a way to directly opening all callers/callees of an element or a way to directly move to the other end of a long connection. 14 participants would like the map to offer filtering features of various kinds. While some would like to be able to hide uninterested parts e.g., all test classes, others would like to filter for special packages in the history view or define time filters like *the last month*. Also on the implementation side there were several suggestions. While some like filters to remove the uninterested elements, others would prefer a more condensed representation or transparency effects to keep the overall context consistent. Another highly requested feature was a quick-search. The idea is to have a text field and while typing only the matching elements are highlighted in the view. Two participants asked for something similar to the metrics information in the outline view of the Tree Rings View to get detailed information for a selected commit or file version. Two think a special coloring for constructors would increase the readability of the Map even more and finally one user asked for a complete online (web) version of the JAVA MAP!

# Conclusion

## 6.1 Conclusion

The overall goal of this thesis was to build a stable and useful version of the JAVA MAP by reaching the point of being a useful aid in program comprehension. This focus was also reflected by the internal working title of the project (JAVA MAP V.2.0). The current version of the JAVA MAP allows the user to analyze a software system on different abstractions levels, starting at the source code, over the internal structures of single classes up to the dependencies between elements over package hierarchies and even projects. The used visual representation is easy to understand and through its tight integration and interactive elements allows the user to quickly access the information needed for the task at hand. While focusing on the element of interest, the whole context is always available if needed. Further more the JAVA MAP offers the user a specialized toolset to access historical information normally hidden in source code repositories.

Finally through the user study conducted at the end of this theses we were able to prove the significant impact of the JAVA MAP on many everyday tasks: A quick introduction of 10 to 15 minutes was enough to empower the participants to use the JAVA MAP efficiently and solve the tasks significantly faster than with the standard tools provided by the ECLIPSE IDE. The feedback we got from all users was highly encouraging and their interest in the tool proves the overall usefulness.

## 6.2 Future Work

As within many software projects, time was the most limiting resource. Although the JAVA MAP is a useful tool today, there is still plenty of room for improvement and enhancement. The longer we worked on the Map and the deeper we dove into the subject, the more we had the feeling that we just scratched the surface of what would be possible. In the next sections we sketch different areas for possible future work.

### 6.2.1 Consolidating the Java Map

Into this section falls all work, which improves the current version of the JAVA MAP without adding any new features to it.

#### **Performance**

As described in Section 5.5, the current version of the Map is considered slow by most users. The targets for this improvement include the analysis, as well as the runtime of the JAVA MAP. We

suggest a in deep performance analysis first and then to use caches and maybe data prefetching and metadata persistence if appropriate.

### **Missing Analyses**

Certain code constructs are currently not supported by the analysis of the JAVA MAP. Static blocks for example, or certain accesses in switch statements are currently ignored during the analysis. To enrich the data model generated by the analysis and to allow further reasoning based on it, this missing parts could be added.

### **Panning of the View**

An intuitive element present in most visual editors except the JAVA MAP! The main challenge in the implementation of this enhancement is to distinguish between a panning and a selection command on a given element.

### **Adding support for other versioning tools**

Currently the JAVA MAP only supports GIT history repositories. To increase the application area of the Map, support for other source code repositories e.g., CVS, Subversion (SVN), Mercurial, etc. could be added.

## **6.2.2 Additional Features**

In this section fall all enhancements, which increase the overall usability, as well as features which open potentially new applications of the JAVA MAP.

### **Increasing the granularity**

Granularity here is meant for both, the analysis, as well as the visualization. Currently the smallest known element by the JAVA MAP is a method or a field. However there is no technical reason for this. We could go one step deeper and investigate the internals of methods as well. This would potentially lead to more and new visual representations placed as an new layer between the code and the current version of the JAVA MAP.

### **Data Flows**

This feature can be seen as an addition to the previous point. Analyzing and displaying the flow of data though a system could be extremely helpful in answering several questions arising frequently from developers as described by Sillito et al. [SMV08]

### **Pattern analysis**

Based on the information of the analysis, pattern recognition mechanism could be implemented. For example, the collected metrics could be analyzed as suggested by Lanza & Marinescu [LM06] and potential matches highlighted in the Map.

### **Filters**

As described in Section 5.5 filter features of various kinds would be highly appreciated by most users and increase the usability of the JAVA MAP even more. This feature includes work in two areas: On one hand we need to define what and how to filter, on the other hand we need to find a consistent visualization form (hiding, transparency or compaction).

### **Runtime Analysis**

Currently the JAVA MAP solely focus on static code analysis. Runtime analysis shows a complete different picture of an application and opens a huge new field. One idea is to add the JAVA MAP



to the debug functionality of the ECLIPSE IDE by allowing the user to navigate through the JAVA MAP at runtime. The information gathered could be used for bug-fixing, as well as input for refactorings and optimizations.

**Quick-Search**

Another highly requested feature by a lot of users is a quick-search. This includes a reasoner (what is the user looking for), as well as a visual representation mechanism (highlighting or fading out).



# Used Tools and frameworks

**Eclipse (Kepler)** - <http://www.eclipse.org>

An open source development environment.

**Eclipse Modeling Framework Project (EMF)** - <http://www.eclipse.org/modeling/emf/>

A modeling framework and code generation facility for building tools and other applications based on a structured data model.

**Graphical Editing Framework (GEF)** - <http://www.eclipse.org/gef/>

A framework to create rich graphical editors and views for the Eclipse Workbench UI.

**Eclipse Memory Analyzer (MAT)** - <http://www.eclipse.org/mat>

An ECLIPSE plug-in used to analyze the memory usage of applications.

**Hibernate** - <http://hibernate.org/>

An object-relational mapping library for Java.

**Teneo** - <http://wiki.eclipse.org/Teneo>

A model-relational mapping and runtime database persistence solution for the Eclipse Modeling Framework (EMF).

**JAVA** - <http://java.sun.com>

An object-oriented and platform independant programming language.

**MigLayout** - <http://www.miglayout.com/>

A Java Layout Manager for Swing, SWT and JavaFX2.

**TextMate2** - <http://macromates.com/>

A  $\text{\LaTeX}$  editor for Mac OS X.

**Gimp** - <http://www.gimp.org/>

A free graphical editing program available on different platforms.

**GIT** - <http://git-scm.com/>

A free & open source, distributed version control system.

**Bitbucket** - <https://bitbucket.org/>

A web-based hosting service for projects that use GIT revision control system.

**R** - <http://www.r-project.org/>

A free software programming language and software environment for statistical computing and graphics, available on different platforms.



# Contents of the CD-ROM

**Zusammenfassung.txt**

German version of the abstract of this thesis

**Abstract.txt**

English version of the abstract of this thesis

**thesis.pdf**

Copy of this thesis

**Java Map.zip**

Eclipse projects containing the source code of the Java Map

**Javadoc.zip**

The javadoc files of the Java Map

**UserStudy.zip**

An archive containing all artifacts of our user study



# User Study

## C.1 Introduction Material

The following material was used during the introduction part of the user study.

## Java Map Introduction

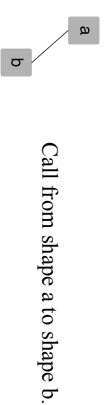
The Java Map is a collection of visualizations, fully integrated into the eclipse IDE. The main goal is to represent elements of a system (code base) in a simplified and abstracted way, to get an easier understanding of the components and their relation.

### Representation

The Java Map is built out of rectangular shapes. The shapes are nested based on parent child relations (i.e., methods contained in a class) and aliened relative to their siblings based on reference relations (i.e., a method is calling another method).

### Layout

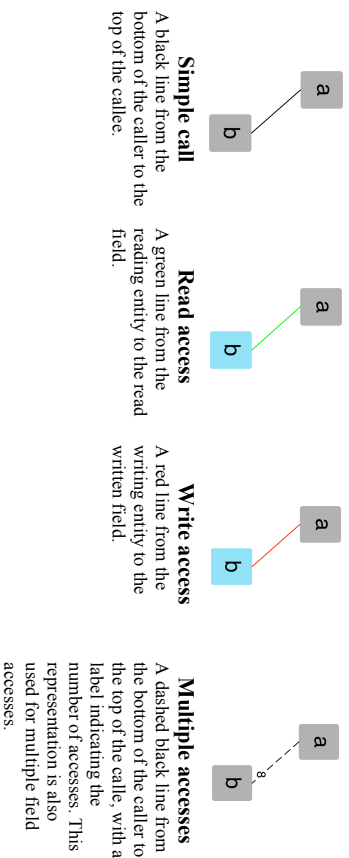
All shapes of the Java Map use the same layout to position their contained children: A called shape is positioned below its caller.



The call relation is represented by a line starting at the bottom of the caller and ending at the top of the callee. This holds for all types of shapes. For example, a class, used (called) by members of another class, is placed below this class. The same holds for package and even projects.

### Connections

There are four types of connections:



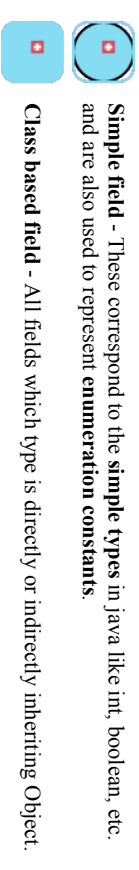
## Shapes

### Simple shapes

Simple or basic shapes represent the smallest entities known by the map.

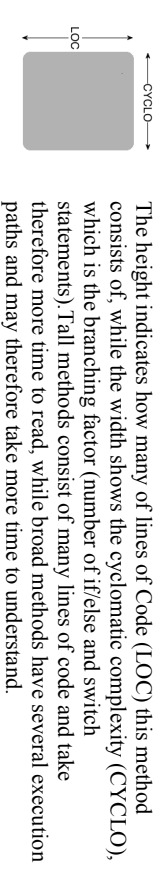
### Fields

The simplest shapes are fields. They come in two versions:

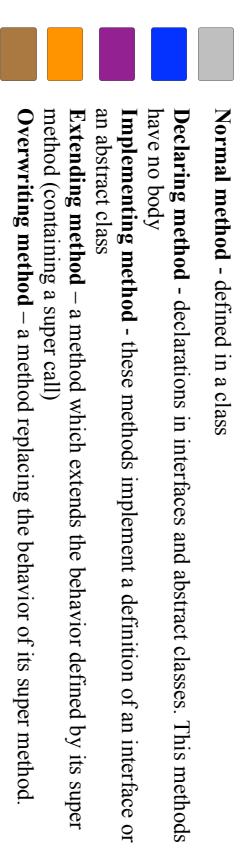


### Methods

Methods are simply boxes with a certain width, height and color:

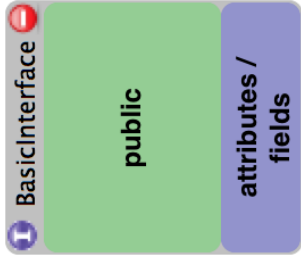


The color simply indicated the inheritance to the super method if one exists. The color codes are like this:

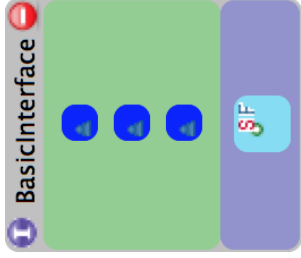




## Interface



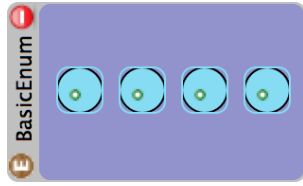
Layers of the interface



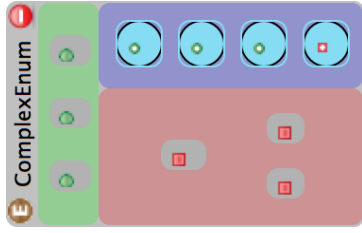
A typical interface

Interfaces consist of two layers. The upper green layer again contains the method definitions, while the lower purple layer holds fields defined in the interface.

## Enumeration



Basic enumeration



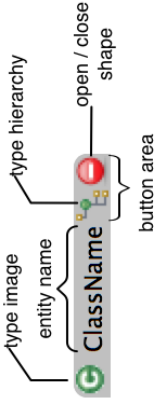
Complex enumeration

Enumerations come in two versions:

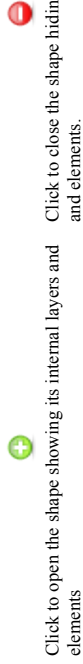
The basic enumeration consist of only one purple layer containing the defined enumeration constants. Because enumerations may become as complex as s classes, the complex enumeration shape looks similar to the class shape.

## Container Shapes

All entities possibly containing other entities (i.e., a class contains methods and fields) are called container shapes. They are represented by a box having a header and several layers. A typical shape header looks like this:



The button area contains controls to get more information for the corresponding shape. The rightmost button is available on every container shape and is used to open or close the corresponding shape:



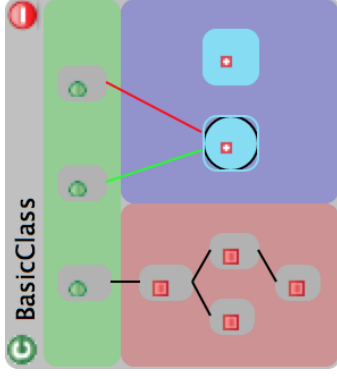
Other buttons like the type hierarchy ( ) are optional and only present if the shape is part of a type hierarchy (i.e., a class having a super class or implementing interfaces)

The next section explains each container shape in detail.

## Class



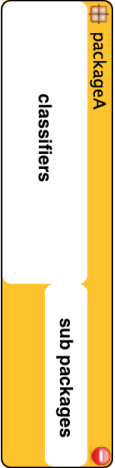
Layers of the class



A typical class

Classes consist of three layers. The green layer at the top contains all public, protected and package private methods and is called the access layer. The red layer on the bottom left contains all private methods and can be seen as the internal mechanics. The purple layer on the bottom right contains all attributes (fields) and inner classifiers (classes and interfaces). Public and private layers use the layout algorithm described under Layout.

Package



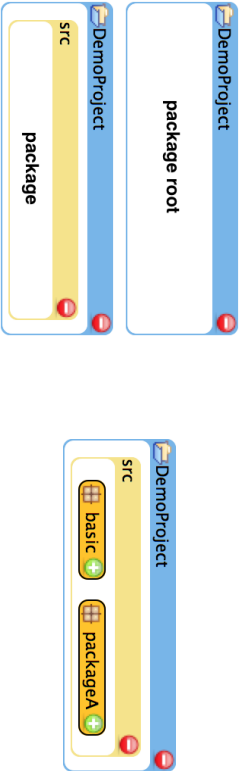
Layers of packages



A typical package

Packages consist of two layers. In the left layer all classifiers (Classes, Interfaces and Enumerations) of this packages are placed. In the right layer all inner (sub) packages are places. Both layers position their children using the layout described under Layout.

Project & Package root



A simple project with one empty package root.

A project with one populated package root.

Project contain only one layer which holds all package roots of the project. Package roots again only have one layer containing the packages. The layers again use the layout described under Layout.

Navigation

Open the Java Map

1. Open a source code editor and place the cursor over the element to display in the map.

2. Press

Outline

The outline view consists of two parts:

Outline

Metric	Value
LOC	122
CYCLO	14
NOM	13
NOC	5
NOP	1
FAN_IN	N/A
FAN_OUT	29

Representation Selection Info 1

In the top part a thumbnail of the current map is displayed. It can be used for orientation and navigation. In the bottom part the metrics of the currently selected element are displayed.

Hovering

Hovering with the mouse over an element will present a small tooltip.

Jumping back to the code

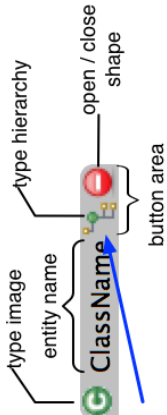
Double clicking on an element (method, field, class, etc.) of the Java Map, opens the source code editor and places the cursor at the corresponding source code element if available.

Zoom

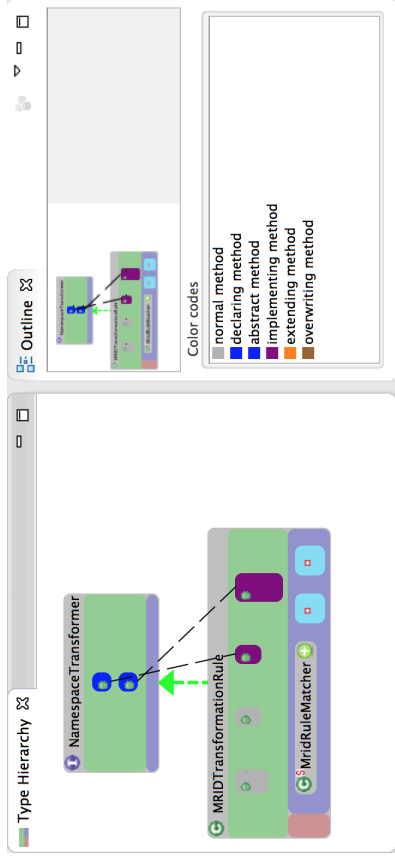
The Java Map provides a basic zoom function, which can be accessed by pressing the command button and using the mouse wheel.

## Inheritance Hierarchy

To analyze the inheritance relations between classes and interfaces of a given project, the Java Map offers the Type Hierarchy View. It is accessible directly from the map. If a shape is part of an inheritance relation, the Type Hierarchy button is shown in the shape header:



Clicking on this button opens the Type Hierarchy View:



In this View the selected classifier (interfaces or classes) together with all super- and subclassifiers is shown. The green dashed arrows indicate the inheritance relation between classifiers, while the black dashed lines indicate method inheritance relations between super- and submethods. The color codes of the methods are the same as for the Java Map.

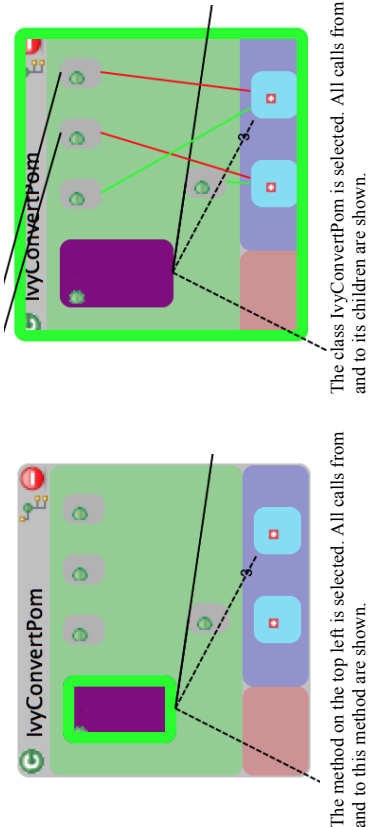
The outline view of the Type Hierarchy view comes again with a thumbnail view for orientation and navigation as well as with a table containing the color codes.

Hovering with the mouse over a certain element will show a popup with the entities name.

The Type Hierarchy View provides a basic zoom function, which can be accessed by pressing the command button  and using the mouse wheel.

## Selection

Selecting an element will reveal the dependency relation in the map. The calls from and to the selected object are shown. If the selected object is a container, all calls from and to its direct children are displayed. Further more the collected metrics of the selected element are displayed in the outline view.



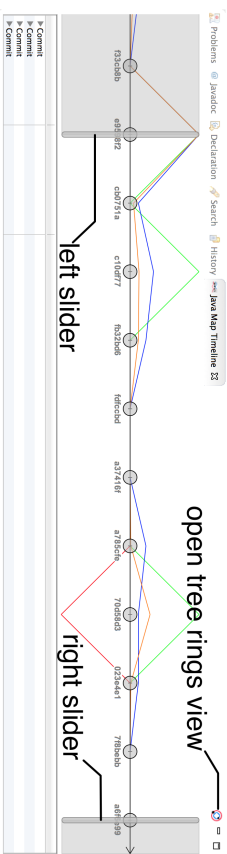
# History

The java Map presents information about the history of a project using two elements:

1. **Timeline** – used to select a time widow of interest
2. **Treerings View** – shows all commits and file versions of the selected time widow.

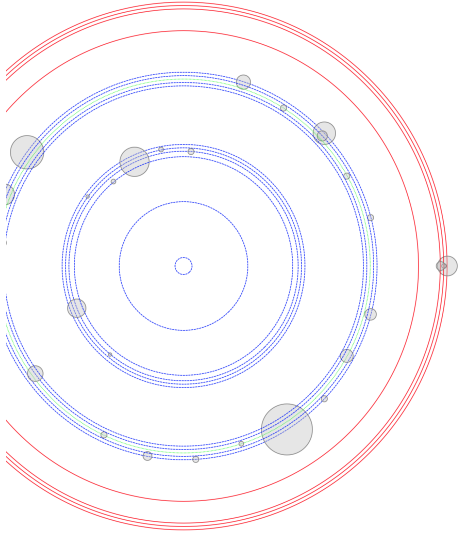
## Timeline

In the timeline all analyzed commits of a project are shown from past (left side) to present right side:



Each commit is represented by a gray circle and label. The graphs indicate the relative amount of changes: Green stands for the amount of added files, red for the amount of removed files, the blue graph indicates how many files have changed and the orange graph indicates how many entities (fields, methods, classes, etc.) have changed in the current commit. Two are used to define left and right end of the time window. Once the time window is set the user can click on the open Treerings View button to show the details of the selected time period.

## Treerings View Representation



Each ring represents one commit. They are ordered from newest in the center to the oldest at the rim of the diagram. The color and line style differ between the authors to make it easier to see who committed what, when. The distance between the circles indicate the relative amount of time passed between the commits: Close rings indicate commits which happened shortly after each other, while larger gaps indicate longer periods without a commit. The gray circles on the rings represent file versions. The size depends on the number of changed entities (fields, methods, classes, etc.). Bigger circles indicate more changes to the same file i.e., several methods of a class were changed. All versions of a file are placed on their commit circle at the same angle. If a file is changed in several commits this leads to a pattern much like a string of pearls.

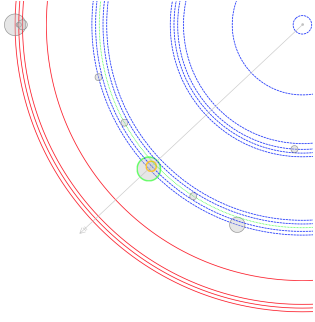
### Navigation

#### Hover

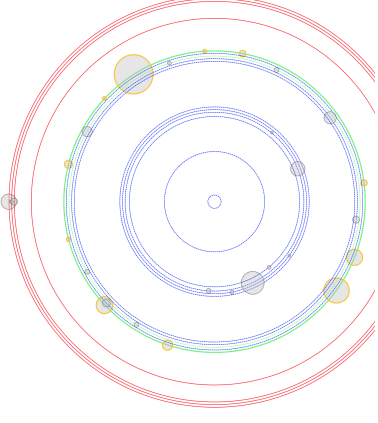
Hovering over a file version or a ring presents a popup with more information.

#### Selection

Selecting a file version will highlight all versions of the file and display an arrow from the center to the rim like a spear indicating the angle on which all file versions lie. Selecting a ring (commit) will highlight all file versions corresponding to it:



The selected file version has a green outline while all other versions of the same file have an orange outline. The arrow indicates the angle on which all versions of the file lie.




The selected ring (commit) has a green outline and all file versions of this commit are highlighted with a orange outline.

### Outline

The outline view offers a thumbnail to orientate and navigate and a list of all authors and their colors of the current time window.

### Zoom

The Treerings View provides a basic zoom function, which can be accessed by pressing the command button  and using the mouse wheel.

## C.2 Questionary

The following pages contain the complete question sets uses for the four participant groups of our study.

## History (Coworkers)

## Questions

- How many authors have worked on the project in the last 10 commits (from cb0751a – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 10 commits)?

Author name

- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 10 commits)? Name the File and the commit.

File	Commit ID

## Type Hierarchy

Analyze the class **LatestVersionMatcher**  
(org.apache.ivy.plugins.version.LatestVersionMatcher)

## Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.

Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

## Entity Access

Analyze the class **JarModule** (org.apache.ivy.tools.analyser.JarModule)

## Questions

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over setter) write access.

Method name

- Which methods of the class JarModule (org.apache.ivy.tools.analyser.JarModule) are called by test classes (classes contained in the package test/java)?

Method name

## Metrics

Please use the spreadsheet to answer the following questions.

## Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.core**.

Answer

- Name the total number of lines of code (LOC) in the class **VersionRangeMatcher** (org.apache.ivy.plugins.version.VersionRangeMatcher)..


Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.trigger**.

Answer

## History (Coworkers)

## Preparation

- Select the Ivy project in the Package Explorer.
- Open the context menu and select “JM” → “Show in Timeline”.
- Adjust the time window by setting the left slider to b6b1f8c and leaving the right slider over a6f6e99.
- Click on the “open Treerings View” (  ) in the top right corner of the Timeline View.

## Questions

- How many authors have worked on the project in the last 15 commits (from b6b1f8c - a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 15 commits)?

Author name





- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 15 commits)? Name the File and the commit.

File	Commit ID

5

## Type Hierarchy

## Preparation

- Open the class **ChainVersionMatcher** (org.apache.ivy.plugins.version.ChainVersionMatcher) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 
- Select the type hierarchy representation for this class by clicking on the type hierarchy symbol (  ) in the top right corner of the class.

## Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.

Method name




- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

6

## Entity Access

## Preparation

- Open the class **Match** (org.apache.ivy.plugins.version.Match) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

**Tip:** Use hover to find the element of interest.

## Questions

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over setter) write access.

Method name

- Which methods of the class Match (org.apache.ivy.plugins.version.Match) are called by test classes (classes contained in the package test/java)?




**Tip:** If you select the package test/java in the map only calls from this package are displayed. If you move back to the class in focus you can see clearly which methods are tested if any.

Method name

7

## Metrics

## Preparation

- Open any class of the Ivy project
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

## Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.plugins**.

**Tip:** To quickly find the package in the Java Map just open one Class of the package and zoom out into the map (cmd + ↓). Now all you have to do is selecting the package to get its metrics.

Answer

- Name the total number of lines of code (LOC) in the class **ResolveReportTest** (org.apache.ivy.core.report.ResolveReportTest).

Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.report**.

Answer

8



History (Coworkers)

Preparation

- Select the Ivy project in the Package Explorer.
- Open the context menu and select “JM” → “Show in Timeline”.
- Adjust the time window by setting the left slider to cb0751a and leaving the right slider over a6f6e99.
- Click on the “open Treerings View” (🔍) in the top right corner of the Timeline View.

Questions

- How many authors have worked on the project in the last 10 commits (from cb0751a – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 10 commits)?




Author name

- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 10 commits)? Name the File and the commit.

File	Commit ID

Type Hierarchy

Preparation

- Open the class **LatestVersionMatcher** (org.apache.ivy.plugins.version.LatestVersionMatcher) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 
- Select the type hierarchy representation for this class by clicking on the type hierarchy symbol (🔗) in the top right corner of the class.

Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.




Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

Entity Access

Preparation

- Open the class **JarModule** (org.apache.ivy.tools.analyser.JarModule) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

**Tip:** Use hover to find the element of interest.

Questions

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over setter) write access.

Method name




- Which methods of the class JarModule (org.apache.ivy.tools.analyser.JarModule) are called by test classes (classes contained in the package test/java)?

**Tip:** If you select the package test/java in the map only calls from this package are displayed. If you move back to the class in focus you can see clearly which methods are tested if any.

Method name

Metrics

Preparation

- Open any class of the Ivy project
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.core**.  
**Tip:** To quickly find the package in the Java Map just open one Class of the package and zoom out into the map (cmd + J). Now all you have to do is selecting the package to get its metrics.

Answer

- Name the total number of lines of code (LOC) in the class **VersionRangeMatcher** (org.apache.ivy.plugins.version.VersionRangeMatcher).

Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.trigger**.

Answer

History (Coworkers)

Questions

- How many authors have worked on the project in the last 15 commits (from b6b1f8c – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 15 commits)?

Author name

- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 15 commits)? Name the File and the commit.

File	Commit ID

Type Hierarchy

Analyze the class **ChainVersionMatcher** (org.apache.ivy.plugins.version.ChainVersionMatcher)

Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.

Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

Entity Access

Analyze the class **Match** (org.apache.ivy.plugins.version.Match).

Questions

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over setter) write access.

Method name

- Which methods of the class Match (org.apache.ivy.plugins.version.Match) are called by test classes (classes contained in the package test/java)?

Method name

Metrics

Please use the spreadsheet to answer the following questions.

Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.plugins**.

Answer

- Name the total number of lines of code (LOC) in the class **ResolveReportTest** (org.apache.ivy.core.report.ResolveReportTest).

Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.report**.

Answer

## History (Coworkers)

## Questions

- How many authors have worked on the project in the last 15 commits (from b6b1f8c – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 15 commits)?

Author name

- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 15 commits)? Name the File and the commit.

File	Commit ID

## Type Hierarchy

Analyze the class **ChainVersionMatcher** (org.apache.ivy.plugins.version.ChainVersionMatcher)

## Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.

Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

## Entity Access

Analyze the class **Match** (org.apache.ivy.plugins.version.Match)

## Questions

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over setter) write access.

Method name

- Which methods of the class Match (org.apache.ivy.plugins.version.Match) are called by test classes (classes contained in the package test/java)?

Method name

## Metrics

Please use the spreadsheet to answer the following questions.

## Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.plugins**.

Answer

- Name the total number of lines of code (LOC) in the class **ResolveReportTest** (org.apache.ivy.core.report.ResolveReportTest).


Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.report**.

Answer

## History (Coworkers)

## Preparation

- Select the Ivy project in the Package Explorer.
- Open the context menu and select “JM” → “Show in Timeline”.
- Adjust the time window by setting the left slider to cb0751a and leaving the right slider over a6f6e99.
- Click on the “open Treerings View” (  ) in the top right corner of the Timeline View.

## Questions

- How many authors have worked on the project in the last 10 commits (from cb0751a – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 10 commits)?

Author name





- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 10 commits)? Name the File and the commit.

File	Commit ID

5

## Type Hierarchy

## Preparation

- Open the class **LatestVersionMatcher** (org.apache.ivy.plugins.version.LatestVersionMatcher) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 
- Select the type hierarchy representation for this class by clicking on the type hierarchy symbol (  ) in the top left corner of the class.

## Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.

Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.




Method name

6

## Entity Access

## Preparation

Open the class **JarModule** (org.apache.ivy.tools.analyser.JarModule) in the source code editor.

- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

**Tip:** Use hover to find the element of interest.

## Questions

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over setter) write access.

Method name

- Which methods of the class JarModule (org.apache.ivy.tools.analyser.JarModule) are called by test classes (classes contained in the package test/java)?




**Tip:** If you select the package test/java in the map only calls from this package are displayed. If you move back to the class in focus you can see clearly which methods are tested if any.

Method name

7

## Metrics

## Preparation

- Open any class of the Ivy project
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

## Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.core**.

**Tip:** To quickly find the package in the Java Map just open one Class of the package and zoom out into the map (cmd + ↓). Now all you have to do is selecting the package to get its metrics.

Answer

- Name the total number of lines of code (LOC) in the class **VersionRangeMatcher** (org.apache.ivy.plugins.version.VersionRangeMatcher).

Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.trigger**.

Answer

8

History (Coworkers)

Preparation

- Select the Ivy project in the Package Explorer.
- Open the context menu and select “JM” ➔ “Show in Timeline”.
- Adjust the time window by setting the left slider to b6b1f8c and leaving the right slider over a6f6e99.
- Click on the “open Treerings View” (🔍) in the top right corner of the Timeline View.

Questions

- How many authors have worked on the project in the last 15 commits (from b6b1f8c – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 15 commits)?




Author name

- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 15 commits)? Name the File and the commit.

File	Commit ID

Type Hierarchy

Preparation

- Open the class **ChainVersionMatcher** (org.apache.ivy.plugins.version.ChainVersionMatcher) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 
- Select the type hierarchy representation for this class by clicking on the type hierarchy symbol (🔍) in the top right corner of the class.

Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.




Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

Entity Access

Preparation

- Open the class **Match** (org.apache.ivy.plugins.version.Match) in the source code editor.
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

**Tip:** Use hover to find the element of interest.

Questions

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “revision” of the class Match (org.apache.ivy.plugins.version.Match), name all methods, which have a direct or indirect (over setter) write access.

Method name




- Which methods of the class Match (org.apache.ivy.plugins.version.Match) are called by test classes (classes contained in the package test/java)?

**Tip:** If you select the package test/java in the map only calls from this package are displayed. If you move back to the class in focus you can see clearly which methods are tested if any.

Method name

Metrics

Preparation

- Open any class of the Ivy project
- Set the cursor to the class name.
- Zoom out into the Java Map using the key shortcut   + 

Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.plugins**.

**Tip:** To quickly find the package in the Java Map just open one Class of the package and zoom out into the map (cmd + ⌵). Now all you have to do is selecting the package to get its metrics.

Answer

- Name the total number of lines of code (LOC) in the class **ResolveReportTest** (org.apache.ivy.core.report.ResolveReportTest).

Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.report**.

Answer

History (Coworkers)

Questions

- How many authors have worked on the project in the last 10 commits (from cb0751a – a6f6e99)? Name them.

Author name

- Which author contributed the most in this timeframe (last 10 commits)?

Author name

- Which file version has the largest number of changes (according to the number of changed methods and fields) in this timeframe (last 10 commits)? Name the File and the commit.

File	Commit ID

Type Hierarchy

Analyze the class **LatestVersionMatcher** (org.apache.ivy.plugins.version.LatestVersionMatcher)

Questions

- Name all methods extending the behavior of their super method.

Method name

- Name all methods overriding the behavior of their super method.

Method name

- Name all methods, which are defined by an abstract superclass or interface and implemented by this class.

Method name

Entity Access

Analyze the class **JarModule** (org.apache.ivy.tools.analyser.JarModule).

Questions

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over getter) read access.

Method name

- For the field “mrid” of the class JarModule (org.apache.ivy.tools.analyser.JarModule), name all methods, which have a direct or indirect (over setter) write access.

Method name

- Which methods of the class JarModule (org.apache.ivy.tools.analyser.JarModule) are called by test classes (classes contained in the package test/java)?

Method name

Metrics

Please use the spreadsheet to answer the following questions.

Questions

- Name the total number of classes (NOC) in the package **org.apache.ivy.core**.

Answer

- Name the total number of lines of code (LOC) in the class **VersionRangeMatcher** (org.apache.ivy.plugins.version.VersionRangeMatcher).

Answer

- Name the number of calls (FAN\_OUT) made by the elements of the package **org.apache.ivy.plugins.trigger**.

Answer

## C.3 Statistical Analysis

For the statistical analysis of our study we used the free software environment R. The following scripts were used to process the collected data:

```
# Load the evaluation data
Evaluation <- read.csv("~/git/java-map-master-thesis/thesis/study/R_Input.
  csv", sep=";")
attach(Evaluation)

# Set current directory
currentDir = "~/git/java-map-master-thesis/thesis/study/images/overview/"
dir.create(file.path(currentDir), showWarnings = FALSE)
setwd(currentDir)

A1_jm = sum(Time[Participant=="A1" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
B1_jm = sum(Time[Participant=="B1" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
C1_jm = sum(Time[Participant=="C1" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
D1_jm = sum(Time[Participant=="D1" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])

A2_jm = sum(Time[Participant=="A2" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
B2_jm = sum(Time[Participant=="B2" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
C2_jm = sum(Time[Participant=="C2" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
D2_jm = sum(Time[Participant=="D2" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])

A3_jm = sum(Time[Participant=="A3" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
B3_jm = sum(Time[Participant=="B3" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
C3_jm = sum(Time[Participant=="C3" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
D3_jm = sum(Time[Participant=="D3" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])

A4_jm = sum(Time[Participant=="A4" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
B4_jm = sum(Time[Participant=="B4" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
C4_jm = sum(Time[Participant=="C4" & Used_JavaMap=="yes" & QuestionID!="H
  .1.3.A" & QuestionID!="H.1.3.B"])
```

```

D4_jm = sum(Time[Participant=="D4" & Used_JavaMap=="yes" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])

A1_njm = sum(Time[Participant=="A1" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
B1_njm = sum(Time[Participant=="B1" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
C1_njm = sum(Time[Participant=="C1" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
D1_njm = sum(Time[Participant=="D1" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])

A2_njm = sum(Time[Participant=="A2" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
B2_njm = sum(Time[Participant=="B2" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
C2_njm = sum(Time[Participant=="C2" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
D2_njm = sum(Time[Participant=="D2" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])

A3_njm = sum(Time[Participant=="A3" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
B3_njm = sum(Time[Participant=="B3" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
C3_njm = sum(Time[Participant=="C3" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
D3_njm = sum(Time[Participant=="D3" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])

A4_njm = sum(Time[Participant=="A4" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
B4_njm = sum(Time[Participant=="B4" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
C4_njm = sum(Time[Participant=="C4" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])
D4_njm = sum(Time[Participant=="D4" & Used_JavaMap=="no" & QuestionID!="H
.1.3.A" & QuestionID!="H.1.3.B"])

jm = c(
  A1_jm, A2_jm, A3_jm, A4_jm,
  B1_jm, B2_jm, B3_jm, B4_jm,
  C1_jm, C2_jm, C3_jm, C4_jm,
  D1_jm, D2_jm, D3_jm, D4_jm)

njm = c(

```



```

A1_njm, A2_njm, A3_njm, A4_njm,
B1_njm, B2_njm, B3_njm, B4_njm,
C1_njm, C2_njm, C3_njm, C4_njm,
D1_njm, D2_njm, C3_njm, D4_njm
)

# print a qq plot for the cumulated time values with the Java Map
fileName = "Q-Q-PlotCumulatedQuestionsWithJavaMap.pdf"
pdf(file=fileName, width=6.78, height=4.32)
#qqnorm(jm, main="Q-Q-Plot of the cumulated questions solved with the Java
  Map")
qqnorm(jm, main="")
qqline(jm)
dev.off()

# print a qq plot for the cumulated time values without the Java Map
fileName = "Q-Q-PlotCumulatedQuestionsWithoutJavaMap.pdf"
pdf(file=fileName, width=6.78, height=4.32)
qqnorm(njm, main="")
qqline(njm)
dev.off()

# print a box plot for the time values
fileName = "Box-PlotCumulatedQuestions.pdf"
pdf(file=fileName, width=6.78, height=4.32)
boxplot(jm, njm, names=c("With Java Map", "Without Java Map"), las=1, ylab="
  Time used in sec.", col="grey")
dev.off()

shapiro.test(jm)
shapiro.test(njm)
t.test(jm, njm, paired=F, alternative="less")
wilcox.test(jm, njm, paired=FALSE, conf.level = 0.99)
t.test(jm, njm, paired=F, conf.level=0.99)

cat("Reduced time needed using the Java Map:")
(sum(njm)-sum(jm))/sum(njm)

jm_median = median(jm)
njm_median = median(njm)

jm_mean = mean(jm)
njm_mean = mean(njm)

jm_sd = sd(jm)
njm_sd = sd(njm)

```

Listing C.1: Calculation of the aggregated values.

```

printQQPlot <- function(fileName, title, data){
  pdf(file=fileName, width=6.78, height=4.32)
  qqnorm(data, main=title)
  qqline(data)
  dev.off()
}

printBoxPlot <- function(fileName, title, jm, njm){
  pdf(file=fileName, width=6.78, height=4.32)
  boxplot(jm, njm, names=c("With Java Map", "Without Java Map"), las=1, ylab="
    Time used in sec.", col="grey")
  dev.off()
}

# Load the evaluation data
Evaluation <- read.csv("~/git/java-map-master-thesis/thesis/study/R_Input.csv",
  sep=";")
attach(Evaluation)

baseDir = "~/git/java-map-master-thesis/thesis/study/images"

questionNames <- c("H_1_1", "H_1_2", "T_2_1", "T_2_2", "T_2_3", "E_3_1", "E_3_2",
  "E_3_3", "M_4_1", "M_4_2", "M_4_3")
questionNamesA <- c("H.1.1.A", "H.1.2.A", "T.2.1.A", "T.2.2.A", "T.2.3.A", "E
  .3.1.A", "E.3.2.A", "E.3.3.A", "M.4.1.A", "M.4.2.A", "M.4.3.A")
questionNamesB <- c("H.1.1.B", "H.1.2.B", "T.2.1.B", "T.2.2.B", "T.2.3.B", "E
  .3.1.B", "E.3.2.B", "E.3.3.B", "M.4.1.B", "M.4.2.B", "M.4.3.B")

p_vals = c()

jm_norm = c()
jm_median = c()
njm_median = c()
jm_sd = c()
jm_min = c()
jm_max = c()

njm_norm = c()
jm_mean = c()
njm_mean = c()
njm_sd = c()
njm_min = c()
njm_max = c()

for(i in 1:length(questionNames)){

  #Create directory for this question if not existent

```

```

dir.create(file.path(baseDir, questionNames[i]), showWarnings = FALSE)
# Set the working directory
setwd(file.path(baseDir, questionNames[i]))

# Extract the completion times for this question with the Java Map
jm = Time[Used_JavaMap=="yes" & (QuestionID==questionNamesA[i] | QuestionID==
questionNamesB[i]) & Time>0]
# Extract the completion times for this question without the Java Map
njm = Time[Used_JavaMap=="no" & (QuestionID==questionNamesA[i] | QuestionID==
questionNamesB[i]) & Time>0]

# print a qq plot for the time values with the Java Map
title = paste(paste("Q-Q-Plot for Question ", questionNames[i], sep=" "), "
with Java Map", sep=" ")
fileName = paste("Q-Q-PlotForQuestion", questionNames[i], "WithJavaMap.pdf",
sep="")
printQQPlot(fileName, title, jm)

jm_norm = c(jm_norm, shapiro.test(jm)$p.value)
jm_median = c(jm_median, median(jm))
jm_mean = c(jm_mean, mean(jm))
jm_sd = c(jm_sd, sd(jm))
jm_min = c(jm_min, min(jm))
jm_max = c(jm_max, max(jm))

# print a qq plot for the time values without the Java Map
title = paste(paste("Q-Q-Plot for Question ", questionNames[i], sep=" "), "
without Java Map", sep=" ")
fileName = paste("Q-Q-PlotForQuestion", questionNames[i], "WithoutJavaMap.pdf"
, sep="")
printQQPlot(fileName, title, njm)

njm_norm = c(njm_norm, shapiro.test(njm)$p.value)
njm_median = c(njm_median, median(njm))
njm_mean = c(njm_mean, mean(njm))
njm_sd = c(njm_sd, sd(njm))
njm_min = c(njm_min, min(njm))
njm_max = c(njm_max, max(njm))

# print a box plot for the time values
title = paste("Box Plot for Question ", questionNames[i], sep=" ")
fileName = paste("BoxPlotForQuestion", questionNames[i], ".pdf", sep="")
printBoxPlot(fileName, title, jm, njm)

w_test = wilcox.test(jm, njm, paired=FALSE, conf.level = 0.99)
p_vals = c(p_vals, w_test$p.value)
}

```

```
result <- data.frame(questionNames)

result["jm_norm"] <- jm_norm
result["jm_min"] <- jm_min
result["jm_max"] <- jm_max
result["jm_mean"] <- jm_mean
result["jm_median"] <- jm_median
result["jm_sd"] <- jm_sd

result["njm_norm"] <- njm_norm
result["njm_min"] <- njm_min
result["njm_max"] <- njm_max
result["njm_mean"] <- njm_mean
result["njm_median"] <- njm_median
result["njm_sd"] <- njm_sd

result["p.value"] <- p_vals

p_adjust = p.adjust(p_vals, method="holm")
result["p.adjust"] <- p_adjust

setwd(baseDir)
write.table(result, file="Result.csv", sep=";")
```

**Listing C.2:** Calculation of the detailed values.

---

# Bibliography

- [dAM08] Brian de Alwis and Gail C. Murphy. Answering conceptual queries with ferret. *ICSE '08. ACM/IEEE 30th International Conference on Software Engineering*, pages 21 – 30, 2008.
- [DL05] S. Ducasse and Michael Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software engineering*, 2005.
- [DLL09] Marco D'Ambros, Michael Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *Software Engineering, IEEE Transactions*, 35:720 – 735, 2009.
- [Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17 – 23, 2000.
- [FM10] Thomas Fritz and Gail C. Murphy. Using information fragments to answer the questions developers ask. *2010 ACM/IEEE 32nd International Conference on Software Engineering*, pages 175 – 184, 2010.
- [GHJ98] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. *International Conference on Software Maintenance*, pages 190 – 198, 1998.
- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
- [Hof08] Marc R. Hoffmann. Eclipse workbench: Using the selection service, 8 2008. <http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html>.
- [KDV07] Andrew J. K, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. *29th International Conference on Software Engineering*, pages 344 – 353, 2007.
- [LM06] Michael Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer, 2006.
- [Mac86] J. D. MacKinlay. Automating the design of graphical presentation of relational information. *ACM Transaction on Graphics*, 5(2):110–141, 1986.
- [MMC02] Jonathan I. Maletic, Andrian Marcus, and Michael L. Collard. A task oriented view of software visualization. *Proceedings of the First International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT'02)*, 2002.
- [OM09] Michael Ogawa and Kwan-Liu Ma. code swarm: A design study in organic software visualization. *Visualization and Computer Graphics, IEEE Transactions*, 15:1097 – 1104, Oktober 2009.

- [PBS93] B. A. Price, R. M. Beacker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, 4(2):211–266, 1993.
- [RC93] G.-C. Roman and K. C. Cox. A taxonomy of program visualization systems. *IEEE Computer*, 26(12):11–24, 1993.
- [RWC12] Dan Rubel, Jaime Wren, and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Addison Wesley, 2012.
- [SBPM08] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF Eclipse Modeling Framework*. Addison Wesley, second edition edition, 2008.
- [Shn96] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. *Proceedings fo IEEE Symposium on Visual Languages*, pages 336 – 343, 1996.
- [SMV08] 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, 2008.
- [Tre11] S. Trentini. Visualizing with evolizer. Bachelor’s thesis, University of Zürich, 2011.
- [Web11] Marc A. Weber. Java map. Bachelor’s thesis, University of Zürich, 2011.
- [Web13] Marc A. Weber. Java map. S.e.a.l. master project report, University of Zürich, 2013.
- [Wet09] Richard Wettel. Visual exploration of large-scale evolving software. *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on 31st International Conference on Software Engineering*, pages 391 – 394, 2009.
- [WL07] Richard Wettel and Michael Lanza. Program comprehension through software habitability. *15th IEEE International Conference on Program Comprehension*, pages 231 – 240, 2007.
- [WLR11] Richard Wettel, Michael Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. *33rd International Conference on Software Engineering*, pages Page(s): 551 – 560, 2011.
- [ZSG79] M. Zelkowitz, A. Shaw, and J. Gannon. *Principles of Soft- ware Engineering and Design*. Prentice Hall, 1979.