Master Thesis

April 1, 2013

Developer Context Model Visualization

Helping Developers to Understand Code by Presenting an Interactive Model of the Developer's Context

Christoph Bräunlich

of Zürich, Switzerland (07-18-722)

supervised by

Prof. Dr. Thomas Fritz

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Master Thesis

Developer Context Model Visualization

Helping Developers to Understand Code by Presenting an Interactive Model of the Developer's Context

Christoph Bräunlich

University of Zurich^UZH

s.e.a.l. software evolution & architecture lab

Master Thesis

Author: Christoph Bräunlich, braeunlich@gmail.com

URL:

Project period: 1.10.2012 - 1.4.2012

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

Acknowledgements

First of all I would like to thank my advisor, Thomas Fritz for his support, advice and for answering my many questions very patiently. I would also like to thank the S.E.A.L.-team, especially Harald Gall for letting me attend various events, which gave me new insights to the work on this thesis and for software engineering research in general. I am also thankful for the valuable feedback and answers I always received promptly from the group members.

Special thanks to my girlfriend Can Liu who has been very patient and supportive during the last six months.

Abstract

Software developers spend a significant amount of their time interpreting and navigating through code. During this process developers build an implicit model of the code structure relevant to the current task they are working on. In an exploratory study we conducted, 68% of the navigation steps of developers working on bug fixing tasks, were revisits. Based on the results of the study, we developed an approach that automatically captures a developer's interaction with code elements in an IDE and presents the relevant code structure graphically to the developer. To evaluate how relevant the elements in the generated diagram are, we simulated the developer's interaction with the IDE that we recorded during the exploratory study. We compared the elements in the diagram after each simulated step of interaction with a chart of relevant code elements that the study subjects drew after completing their tasks. The results show that in the simulations the elements which the study subject revisited were visible in the generated diagram before the re-visitation happened. From this finding, we can imply that the prototype has a potential to save time in software change tasks. Finally, we propose improvements to the system based on our evaluation.

Zusammenfassung

Software Entwickler setzen einen signifikanten Teil ihrer Zeit zum Interpretieren und Navigieren durch Quellcode ein. Für ihre aktuelle Aufgabe bilden die Entwickler ein implizites Model der Code-Struktur. In einer eigenen explorativen Studie mit Entwicklern, die an Fehlerbehebungsaufgaben arbeiteten, führeten 68% aller Schritte zu Code-Elementen, die der Entwickler bereits inspiziert hatte. Basierend auf dieser Studie entwickelten wir einen Prototypen, der automatisch die Interaktion zwischen Entwickler und Entwicklungsumgebung registriert und die also relevant eingestuften Code Elemente graphisch darstellt. Zur Untersuchung der Frage, ob die als relevant kategorisierten Code-Elemente für Entwickler effektiv als relevant angesehen werden, simulierten wir die Interaktionen, die während der Studie aufgezeichnet wurden. Wir verglichen die die Elemente im automatisch erstellten Diagramm mit Elementen aus den Diagrammen, die die Studienteilnehmer zeichneten, nachdem sie ihre Aufgabe erfüllt hatten. Die Resultate der Evaluation zeigten, dass die Code Elemente, die mehrfach besucht wurden in der Simulation vor dem Wiederholten Besuch angezeigt. Aus diesem Befund lässt sich ein Potenzial ableiten, dass mit diesem Ansatz in Aufgaben für Programmänerungen Effizienzsteigerungen erziehlt werden können. Die Arbeit schliesst mit einer Analyse des entwickelten Prototypen und schlägt Verbesserungen vor.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

Nowadays, software development is usually followed by a long phase of maintenance, which lasts through the whole life time of a software. During this phase, developers spend a significant amount of time navigating through unfamiliar code [KMCA06, SLVA10]. The code is often not structured according to the context of a maintenance task. Instead, it is modularized according to other structures that are defined during the development phase. Although there exist attempts to organize code with the notion of aspects according to specific tasks [KLM$^+$97]. But usually, developers have to depend on their own exploration to understand the code. They often start with searching for relevant code elements and explore the dependencies of them. With the discovered information about the relevant code, developers build a implicit model to remember what was found [KMCA06, MKRČ05]. This whole process might take more time than actually completing the maintenance task [KM05]. Developers might also fail in finding the relevant code if they do not follow a methodical approach [RCM04].

Some researchers have addressed these issues and have built tools to help developers focus on a given task. *Code Bubbles* [BRZ$^+$10] lets developers organize the code by placing *bubbles* on a large virtual canvas. A bubble may contain any task-related information such as the code of a method or relevant documentation. *Code Bubbles* also allows developers to save and share the set of *bubbles* related to a task so that other developers may work on it directly instead of building up a new workspace. *Relo* [SKM06] follows a different approach to help developers explore the code structure. It displays an interactive UML class diagram, which shows relevant code elements and hide irrelevant ones. *Mylyn's* approach was proved to be very successful and became a part of the Eclipse IDE. It tracks the developer's interaction with the code and automatically learns which elements are important to the developer. The code elements categorized as relevant are highlighted in the Eclipse Package Explorer, whereas irrelevant code elements are hidden. In *Mylyn*, the generated model can be assigned to a task in an issue-tracking system and shared in a team.

In this thesis we propose a new approach that combines and extends several existing approaches. An exploratory study was conducted, where several developers were asked to work on a software-change task and draw a visual representation of their mental model. We transcribed the developers' navigation and analyzed the collected data. Based on this study we built a prototype that captures the developer's navigation within the code and automatically assigns a relevancy-value to each code element in this navigation model. Combined with a static code analysis, it generates an interactive UML class digram-like chart, which visualizes the structure and relation of the most relevant elements. The chart is designed to be similar to the charts drawn by the participants in the study. Developers are also able to see the source code assigned to the diagram by clicking on code elements in the diagram. The approach was implemented as a Eclipse plug-in and can thus be embedded in existing development environments.

1.2 Research Contribution

The main contribution of this thesis is an approach to automatically analyze the developer's interaction with an IDE and generate a graphical representation of a context specific model. It helps developers build an implicit model of the code elements that are relevant to a specific task. We also show how to extend the presented prototype to have a more constant chart over time and how to add additional structural information to the diagram. In particular we wanted to address the following questions:

1. How can a system help developers to find relevant code elements and their relations in unfamiliar source code for a specific task?
2. How can this relevant code and structure be visualized to support the developer to build an implicit model?

1.3 Structure of the Thesis

In this thesis, we start with presenting an exploratory study on developers' code navigation, which leads to several conclusions. They are used to design the approach to help developers comprehend the source code faster while navigating. we continue with demonstrating a prototypical IDE plug-in that implements the approach. Afterwards we describe the evaluation of the prototype based on the data gathered in a study. In the end, we discuss how the prototype could be extended in future work.

Chapter 2

Related Work

The related work can be divided into three categories. First, studies that investigate how developers find the information they need to work with code and how they build and implicit model of this information (Program Comprehension). Secondly tools, often based on the program comprehension studies, that find the relevant code elements for the developers to help them understand the code faster (Code Recommendation). The last category lists tools and methods that help developers navigate through the code by presenting them the the relevant code elements structured in a graphical representation (Navigation Support).

2.1 Program Comprehension

In two similar studies [KAM05,KMCA06], Andrew Ko et al. described an exploratory study with which the tried to explore what software developers in a software change task do, until they understand the code well enough to finally perform the change in the software. Ko et al. laid a focus on how developers decide what is relevant for them, what types of relevant information they seek, how they keep track of the gathered information and what developers do differently than other developers for the same task. They fond out that developers spend in average 35% of the time navigating through code that was relevant for the task and developers use 46% of their time going through irrelevant code. They also show that the purpose for most re-visitations of code was juxtaposing code fragments.

In an other study [LGHM07], ten novice developers and three expert developers were given three hours to understand the design of an open source project and to find problems in the design. The study shows that the expert developers were able to comprehend the root cause of the design problems while the novice developers could only explain the symptoms. During the analysis, the experts did not read more relevant methods than the novices but they also did not waste a lot of time on irrelevant methods.

In [LM10], the authors define the term **Reachability Question** and explain it as follows:

> A *Reachability Question* is a search across feasible paths through a program for target statements matching search criteria.

In their paper, the authors showed that in three studies developers had to ask reachability questions very often (e.g. in one study 9 times a day). Typical questions were about which code elements are impacted by a specific change of source code. The authors further state that available tools that present call graphs to the developers are not able to show all affected code elements and tools for that purpose have yet to be developed.

One Year later, the same authors present such a tool called **Reacher** in [LM11]. This tool presents an interactive call chain graphically to the developer. With a statement matcher integrated into the tool is it possible to find elements affected by a change that can not be seen with traditional call graph tools, often seen as part of IDEs. In a study the authors show that *Reacher* helps developers exploring code more easily and more effective.

A study about the costs and benefits of UML in software maintenance [DAB08] shows that UML helps developers to understand code better and if used is beneficial for the functional correctness of changes. However to generate and continuously update the documentation in form of UML diagrams makes the development more time-consuming.

2.2 Code Recommendation

The Eclipse plug-in **Mylyn** [KM05] solved the problem of information overload with the addition of the notion *task* to the IDE. With help of the user interaction it creates Degree-of-Interest model for the current task with three simple rules that are applied while the developer navigates through code. The generated Degree-of-Interest model can then be used to filter irrelevant files and adds a degree-of-interest-value to code elements. In 2006 Mylyn has been extended with an ability to associate the Degree-of-Interest values of code elements to tasks [KM06] and now allows developers to switch between tasks. Mylyn is well integrated into Eclipse and other task management tools like Bugzilla [bug]. With help of these tools, the task can be shared between developers.

Sando [SDRF12] is a framework for experimenting and quick integration of several information-retrieval approaches to find relevant code elements to a given search term. The *Sando search tool* was developed on top of the framework to demonstrate the approach. Based on the same study as used in this thesis, the authors showed that a following tool **CoMoGen** [FSB12] that uses a similar but refined approach performs better than *Sando* and that both *Sando* and *CoMoGen* yield significantly better results than a normal text search.

2.3 Navigation Support

Relo [SKM06] is a tool to help developers to understand parts of large code bases with help of an interactive UML-like diagram that shows more information when the developer clicks on buttons inside the diagram. In this tool the mental model of the developer is rebuilt by the developer himself within the diagram. It is also possible to show the diagram after code navigation in the source code editor, the diagram then shows all previous navigation or the manually filtered navigation. The filtering can be done with the help of a designated dialog before showing the diagram. The tool includes an embedded text editor below the visualized code elements which allows the developer to use the tool as a single point of interaction with the IDE. Since the tool only shows the part of the code base which was explored by the developer, it reduces the cognitive overload. Relo was turned into a commercial product under the name Architexa [arc].

The authors of **Code Bubbles** [BRZ$^+$10], propose a new user interface for IDEs that displays methods separated in a small window which they call *bubble* on the screen. Developers can navigate with the help of these bubbles and open new related methods in new bubbles. With that method, the implicit model that developers build in their heads can be represented on the display and additionally annotated. Code bubbles also has advanced built in debugging support and can automatically display a call graph between to methods. It also allows users to add notes and add flags (icons) to bubbles. To overcome the problem of not having enough screen real estate, the code bubbles are arranged on a large virtual space which can be panned by the developer. A set of bubbles can be saved in a session and shared within a group of developers by emailing a xml session file.

Stacksplorer [KKD^{+}11] supports developers comprehending source code by presenting a list of methods that call a currently selected method and another list of methods that are being called by the currently selected method. By clicking on the methods the tool allows the developer to interactively browse the automatically computed call graph. In a study, the authors of the paper show that tasks in a large open-source project could be completed significantly faster with the use of *Stacksplorer*.

Concern Graphs [RM02] is a representation of code elements that belong to a specific concern of a developer in a project. In their paper, the authors present the Feature Exploration and Analysis Tool (FEAT) [RM03], which shows the concern graph to the developer in a tree structure. In three case studies, the authors show that concern graphs allow developers to focus on critical parts of the program, that the concern can be displayed even if it is scattered across the whole program and that concern graph scale up to large systems.

Chapter 3

Exploratory Study on Developer Navigation

Thomas Fritz and David Shepherd, Industrial Software Engineering Researcher at ABB's Software Engineering Research Group, conducted an exploratory study [FSB12] in which they observed 12 software developers completing a change task in one of three open source projects [fre], [jpa], [rac]. While the developers investigated the code, their screen was recorded. The resulting screen recordings were later used to analyze the steps the developers performed in order to complete the change task. Out of these transcriptions a *Navigation Model* was created for each developer. The developers were also asked to draw a diagram of their mental model of the code that is relevant for the task. For this models we will use the term *Developer Model* later in this thesis. After the developers completed their change task, the code changes were analyzed and all changed code elements were listed in a *Patch Model*.

The author of this thesis performed most of the transcription and a big part of the data analysis. All transcripts as well as developer and context models collected during the study are available at [stua].

3.1 Study Design

3.1.1 Research Questions

With the study, the authors wanted to get a better understanding of code context models and based on what insights software developers build these models i.e. which parts of the code are relevant for a developer and should therefore be part of the model. The following questions that are intended to be answered by the study are relevant to this thesis:

- What is a developer's mental model of the source code after completing a given task?
- Which code elements are relevant in a certain context of a change task?
- Can developers keep the relevant elements in mind?
- How do people navigate? And how often do they use the structural navigation aids provided by an IDE?
- Is it more effective to navigate using the structural navigation aids from an IDE?

3.1.2 Study Methods

Three open source projects were chosen which should show recent development activity and systems that are big enough so that developers can not understand the entire system in a systematic way. The projects should also have a open bug reporting system with a task that can be completed within less than an hour and they should all use Java as programing language. Specifically the following projects were chosen: FreeMind [fre], a mind-mapping software, Java Password-Safe [jpa], a software to store and encrypt passwords, [rac], a software to timetrack projects. 10 of the 12 developers completed their tasks successfully.

Transcribing the Developer's Navigation

To have a similar development environment, all developers connected themselves to a remote desktop with Eclipse already setup for the task. The screens were recorded during the work on the task so that all steps could later be analyzed. In order to do so we systematically analyzed the resulting videos and transcribed all relevant steps into a table for each developer. All transcriptions together with the other study artifacts can be found on-line [stub].

The transcriptions were done by the author of this thesis and cross validated by Professor Thomas Fritz. The resulting transcripts are used in both the referenced paper [FSB12] and in this thesis, which were partly written at the same time.

Action	Time	Element Name	Type of Element	Containing Type	Type of Relation	Category	Input Method	View	Notes
					...				
Open Type	07:51	ControllerAdapter.SaveAction	Class	ControllerAdapter	Definition	Navigation	Keyboard	Java Editor	Ctrl+Shift+T, "free*.S*Action"
Scan	07:51	ControllerAdapter.SaveAction	Class	ControllerAdapter	Definition	Navigation	Mouse	Java Editor	
Scan	08:03	FreeMind_integration	Project	-	-	Navigation	Mouse	Package Explorer	
View Task	08:07	Task	Mylin Task	-	-	View	Mouse	Mylyn Window	
Open Type	08:16	ControllerAdapter.SaveAction	Class	ControllerAdapter	-	Navigation	Mouse	Java Editor	Click on Java Editor Tab
					...				
Quick Hierarchy	09:46	save(File)	Method	MapAdapter	Definition	View	Keyboard	Java Editor	Ctrl+T
Show Method	09:54	save	Method	MindMapMapModel	Definition	Navigation	Mouse	Java Editor	Using Quick Hierarchy
Show Method	09:58	saveInternal	Method	MindMapMapModel	Reference	Navigation	Mouse	Java Editor	Ctrl+Click on Reference
					...				

Table 3.1: Sample of the transcript for subject F1

Since the subjects were using the Eclipse IDE we tried to use Eclipse's notation whenever possible in the transcript. If there existed an action that we wanted to transcribe named in Eclipse, we used this name. For instance we used "Open Type" whenever a Class or Interface was opened in a Java Editor because of the *Open Type Dialog* in Eclipse.

Navigation Model

To get a better understanding about how the developers navigated through the code we extracted all navigation actions from the transcripts (a sample of the transcript for subject 9 is shown in Table 3.1). In this analysis, non-navigation actions were omitted e.g. the fourth in Table 3.1 - Viewing the task which the developer had to perform in Eclipse's Mylyn window.

We categorized all navigation steps in *Structured* ones and *Unstructured* ones. *Structured Navigation Steps* include navigation aids like *call hierarchy*, *type hierarchy* and *find references*, *debugging steps*: *step into*, *step return* and *stacktrace clicks*. Finally Structured Navigation Steps also include navigation steps aided by extended editor functions like *quick documentation*, *open declaration* and *quick fix* (which was only used in the FreeMind project to open the parent class). We categorized the following steps as *Unstructured navigation steps*: expanding and opening items in the package explorer, performing Java and File searches, finding string in a file and using the outline view to navigate as well as navigation forward and backwards in Eclipse's navigation history, open editor tabs and scanning code.

A visualized version of subject F1's navigation model (Figure 3.1) shows how the developer often returned to the same elements. For example the methods *MindMapMapModel.save* and *MindMapMapModel.saveInternal* stood in the focus of the developer in several navigation steps. With only 40.0% steps that navigated to a previously inspected element, Subject F1 performed less revisits than the average developer that completed the task successfully (in average 68.0% of all navigation steps are revisits). We choose to show this picture because the graph is compact. Some of the graphs are more scattered and would require several pages to be displayed while other graphs are to simple to show. Most of the revisits happened during the initial phase of comprehending the code.

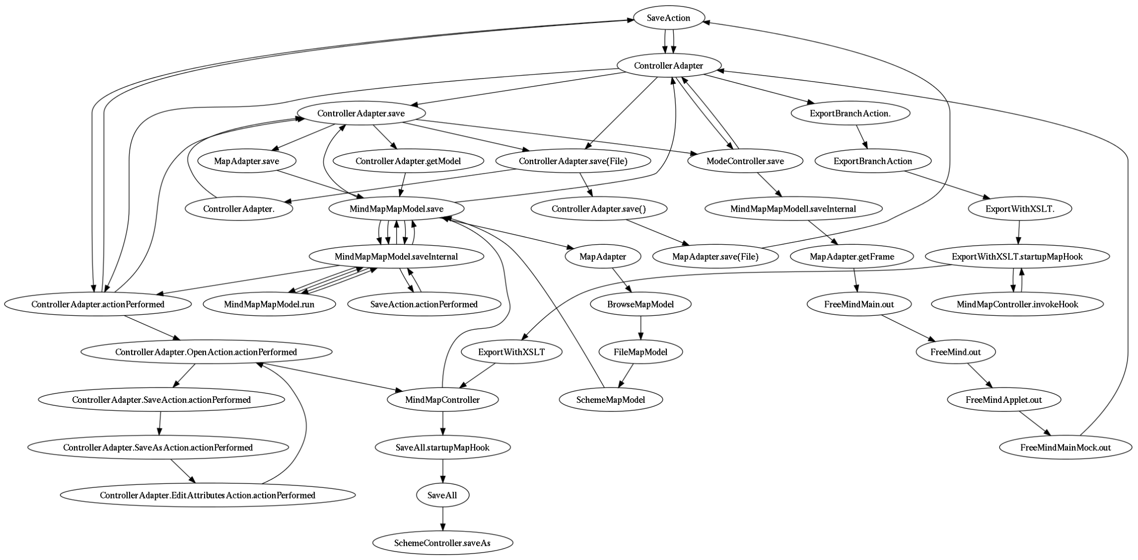


Figure 3.1: The visualized navigation model of subject F1 shows that the developer revisited some code elements like *MindMapMapModel.save*, *MindMapMapModel.saveInternal* or *ControllerAdapter* several times.

Developer Model

After the developers completed their task, in a questionnaire they were asked what information they needed to understand in order to complete their tasks. More specifically they were asked which software artifacts and source code elements were relevant to them and which relation between those code elements had an impact on their understanding of the context of their tasks. The developers then were asked to draw their mental models using pen and paper or a computer drawing tool as a diagram in any form (See two examples in Figure 3.2).

Some of these diagrams were drawn similar to a UML class diagram, others were more abstract (like the picture in on the right in Figure 3.2). To make these models comparable, we listed all code elements from the diagrams. In this process some of the information certainly got lost but since the models were different in therms of the chosen abstractness, it seemed like a valid method to equalize the models. To eliminate ambiguities and errors in the drawings, we compared the extracted elements with the source code.

Of the 12 diagrams only 7 contain methods. The average number of classes in the diagrams is

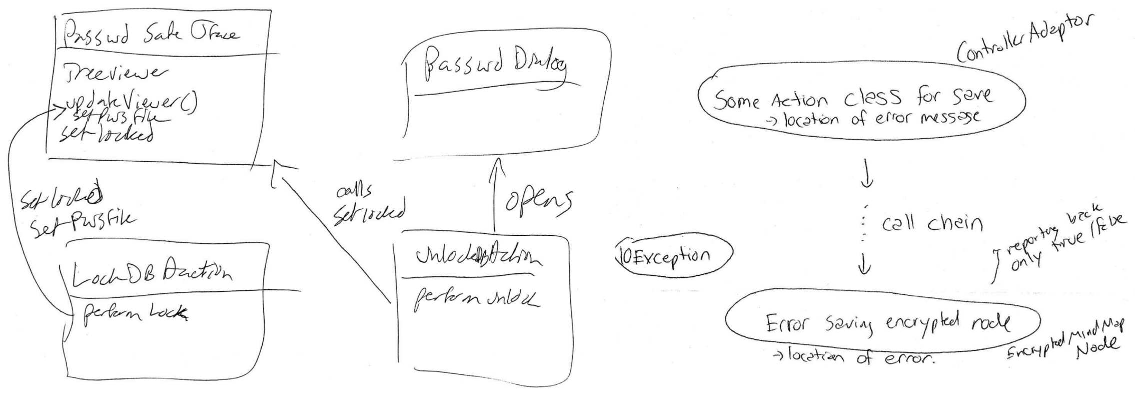


Figure 3.2: Developer models drawn by subjects J1 and F2 after completing their tasks. The developer model of J1 uses a UML class diagram-like notation whereas F2 uses a more abstract graphic. Both models explicitly refer to code elements.

Classes Subject J1	**Classes Subject F2**
org.pwsafe.passwordsafeswt.PasswordSafeJFace	freemind.modes.ControllerAdapter
org.pwsafe.passwordsafeswt.action.LockDbAction	freemind.modes.mindmapmode.EncryptedMindMapNode
org.pwsafe.passwordsafeswt.action.UnlockDbAction	freemind.modes.ControllerAdapter.SaveAction
org.pwsafe.passwordsafej.PasswordDialog	java.io.IOException
Methods Subject J1	Subject F2 has no classes in his model
org.pwsafe.passwordsafeswt.PasswordSafeJFace.updateViewers() : void	
org.pwsafe.passwordsafeswt.PasswordSafeJFace.setPwsFile(PwsFile) : void	
org.pwsafe.passwordsafeswt.PasswordSafeJFace.setLocked(boolean) : void	
org.pwsafe.passwordsafeswt.action.UnlockDbAction.performUnlock() : boolean	
org.pwsafe.passwordsafeswt.action.LockDbAction.performLock() : void	
Fields Subject J1	Subject F2 has no fields in his model
org.pwsafe.passwordsafeswt.PasswordSafeJFace.treeViewer : TreeViewer	

Table 3.2: Code elements extracted from the developer models of subjects J1 and F2. Even though F2 refers to a call chain, no methods or fields could be extracted from F2's developer model.

4.1 and the average number of methods in the diagrams that contain methods is 3.86.

3.1.3 Subjects

An overview of the study participants can be seen in Table 3.3. The 12 subjects were recruited from the industry (six professional developers) and from academia (three graduate students and three faculty members). All participants had at least 8 years of experience in software development with an average of 11.83 years with an average of 51.84 minutes.

3.2 Study Results

10 of the 12 study participants completed the task successfully. The successful developers completed their tasks in a time between 8.6 and 100.6 minutes. In average the developers needed 258.3 ($stdev = 159.7$) navigation and debugging steps for the task of which 27.83 ($stdev = 20.66$) were

steps aided by structural navigation tools from the Eclipse IDE (e.g. *follow type hierarchy*) and 88.7 ($stdev = 88.24$) debugging steps (like *step into*). Of the 258.3 steps they needed in average 174.2 ($stdev = 130.08$) steps navigated the developer to a code element that was already previously visited (listed as *Revisits* in Table 3.3).

Project	*ID*	*Job*	*Years Pr.Exp.*	*Time & Success*	*Dev. Model*	*Patch Model*		*Code Nav. Model*		*Navigation Steps*			
					Cl	*Cl*	*Me*	*Cl*	*Me*	*All*	*Structured*	*Revisits*	*Deb*
Freemind	**F1**	pro	10	39.7min ✓	4	16	25	37	42	116	101	47	26
	F2	pro	11	22.0min ✓	4	15	23	16	25	106	57	40	54
	F3	grad	8	59.7min ✓	5	11	18	19	36	341	229	250	219
	F4	grad	9	70.9min ✓	4	16	23	22	38	177	41	112	11
JPass	**J1**	pro	12	8.6min ✓	4	2	2	6	12	67	46	30	33
	J2	pro	12	64.5min ✓	4	6	11	19	28	408	279	305	250
	J3	pro	11	62.0min ✓	5	3	7	12	27	140	64	88	17
	J4	fac	12	101.1min ■	4	-	-	19	32	570	459	453	441
Rachota	**R1**	fac	15	53.8min ✓	6	5	10	20	31	349	101	248	78
	R2	grad	11	100.6min ✓	4	1	2	13	78	553	42	396	39
	R3	pro	15	36.6min ✓	6	1	2	10	22	326	130	241	160
	R4	fac	16	114.4min ■	9	-	-	16	35	282	76	155	6

Table 3.3: This table from the report [FSB12] presents a summary of the descriptive statistics collected on the developer's background and from the exploratory study ($pro = professional$, $grad = graduatestudent$, $fac = faculty$, $\checkmark = success$, $\blacksquare = failure$, $Cl = classes$, $Me = methods$, $Deb = debugging$).

The study results that are relevant for this thesis can be summarized as follows:

Developers often revisit code. Having a re-visitation rate of 61.4% ($SD = 14.7\%$), it seems that the mental model of a developer has to be extended or can not be kept in mind. Often, revisits are short. Presumably they only remind themselves of small things, like structures or names of methods they want to call. Not surprisingly, developers with a higher re-visitation rate spent more time to complete the whole task (Pearson's $r = 0.72$).

Developers that use the structured navigation aids of the IDE can complete their task faster. With a Pearson product-moment correlation coefficient of $r = -0.49$, we support the observation of *Robillard et al.* [RCM04] that successful developers perform more structurally guided searches than unsuccessful ones. However, in the exploratory study, we looked at the task completion time rather than on the success.

Code Navigation Models are highly connected. As can be seen in figure 3.1, the navigation models are often (six of the ten navigation models, of the tasks that were completed successfully) clustered around a one highly connected group of elements (in this case around MindMapMapModel). In average, 73% of all classes were connected to at least one other class.

Developer Models are small. When developers are asked to draw the basic structure of the model they have in mind, they will come up with only few elements which are relevant to the task. In average the developer models include 6.3 elements of which 4.1 are classes, 2.3 are methods and only 0.9 are fields.

3.3 Contributions of the Author

The author of this thesis transcribed the screen recordings that were taken when the developers solved their change tasks. The author also analyzed the drawn developer models and extracted the code elements for further analysis. Then he analyzed the transcripts and classified the all navigation steps into the following categories:

- Structure Navigation
  - Follow Dynamic Program Flow
  - Follow Declarations
  - Follow References
  - Follow Type Hierarchy
- Text Searches
- Scans
- Unstructured Navigation
- Debugging
- Code Editing

For the categories of steps he calculated the ratios and correlation to the time the developers needed to complete the tasks. The results of most correlations confirm results of other studies (e.g. [RCM04]). The correlations lead to the following conclusions:

Less revisits led to a faster completion of the task, regardless of how many navigation steps were performed. With a correlation coefficient of 0.62 between the number of revisits and the time to complete the task, developers that revisit the same code less were able to complete their tasks faster. Probably, this is because more advanced developers have to revisit less and can complete the task faster.

Developers that debugged more were slightly more productive but programmers that often debugged the same code were less productive Some developers in the study seemed to use the debugger very target-oriented to find dynamic call chains this increased the productivity. However, developers that used the debugger to analyze all code and tried to perceive the whole structure with this tool mostly needed more time than developers that read most of the source code without the help of a debugger. (Correlation coefficient between the ratio of the debugging and other steps and the total time: 0.44)

Unstructured navigation slightly decreased productivity and **Developers that followed Type Hierarchies more often typically needed less time.** Following type hierarchies was one of the more frequently used structured navigation aid in the study. We assume that developers that used aids like this were specifically inspecting inheritances to other structural dependencies and could therefore build a implicit model faster. (Correlation Coefficient between the ratio of the structured and unstructured steps and the total time: 0.34)

Developers that used the debugger more and spent more time on editing code were more productive Assuming an implicit model and trying to confirm this model my editing code and debugging seems to be a technique which corrects and builds an implicit model quickly. The correlation coefficient of 0.55 is relatively small, we were still able to draw the conclusion because one developer that was very slow used the debugger almost exclusively and reduced the correlation coefficient noticeably.

Chapter 4

Approach

The findings of the exploratory study show that developers can not keep all relevant code elements in mind. Presumably it is not only the structure that makes the developers revisit the same code elements again but personal experience suggests that the structure of the code is relevant when locating a bug in a larger code base. There already exist several tools that show the structure of the source code to the developer (e.g. Relo [SKM06]) but the developers have to explore the diagram themselves. Developers are used to browse through the code and associating the code with a model they have in mind because they have to work with code on a daily basis. But the UML class diagram as used in Relo is not used in every software project [DP06] even though most developers have heard about it and know the structure from books and documentations. Therefore we propose the use of the code to UML class diagram association the other way around: software should work directly with the code as they are used to and a UML class diagram (or something similar) should be generated automatically as a documentation of the relevant code elements and the structure that connects the elements. To help the developer to associate the elements drawn in the diagram to the code elements in the source code editor and to help developers understanding the underlying code, the diagram should be interactive.

Approaches to capture relevant code elements during code navigation have already proven to be usable like Mylyn [myl, KM05] that became a standard component of the Eclipse IDE. Mylyn determines the relevant code elements with a degree-of-interest model and highlights important classes in the package explorer while unimportant ones are hidden. In our approach we want to extend Mylyn's degree-of-interest model and use it to generate an interactive UML class diagram-like representation of the code relevant to a specific task. To better show the structure of the code elements, the underlying code should be statically analyzed to determine and later visualize the relations between the relevant code elements.

4.1 Heuristics and Static Code Analysis

Mylyn's degree-of-interest model uses code edits and code selections to increase the interest-value of the corresponding code elements. While the interest-value raises for the affected elements, other elements loose in interest. The approach presented here extends Mylyn's degree-of-interest model by additional heuristics to determine the relevance of code elements and adds static code analysis to determine the relations between code elements. Our approach assigns a relevance-value to each Java type, Java method and Java field. It further defines three thresholds for the relevance values:

- The **Relevancy Threshold** defines whether an element is relevant enough to be displayed in the diagram. I.e. a code element is categorized as relevant if its relevance-value is higher or equal than the relevancy threshold.

- The **Importance Threshold** determines whether a code element is especially important and should be highlighted in the diagram. Code elements with a relevancy-value higher or equal to the importance threshold are categorized as important.
- A **Minimal Relevancy** does not allow an element to have a relevancy-value which is that low, that the element can never become relevant in a common task solving programming session anymore.

The heuristics that determine the relevance-values of the code elements should be designed to be expendable and the following list should not be considered complete in a sense that it generates an optimal result for various different tasks. The following heuristics were generated out of the insights of the exploratory study and should yield good results for the same tasks that were analyzed in the study:

- The relevancy threshold is set to 0.
- The importance threshold is set to 2.
- The minimal relevancy it set to -0.9.
- Selecting a code element increases the relevance by 1.1.
- Selecting a code element decreases the relevance of all elements in the model by 0.1.
- Selecting a code element marks the element as active and sets the state of the previously marked as not active anymore. Active code elements are highlighted in the diagram to indicate to the developer which code element was selected in the last step and which code element is therefore most likely to be currently visible in the code.
- Whenever a Java method becomes active, all methods that call the methods and all methods that are being called by the active method become visible in the diagram. An arrow from each caller to the active method and an arrow for each callee from the active methods form a call chain in the diagram.
- Whenever a code element becomes relevant, the structural dependencies of that element are determined with a static code analyzer.
- All classes that include a relevant code element are also considered relevant even if they have a relevancy-value below the relevancy threshold.

4.2 Visualization

The approach is meant as an extension for IDEs and the visualized diagram should be an addition to components already available to an IDE like source code editors.

Figure 4.1 shows the concepts of the visualization. The approach uses a UML-class-diagram-like notation which additionally makes use of icons for the code elements like this is done for code elements in the Eclipse IDE. It simplifies relations between code elements by abstracting all relations that can occur in the diagram into three categories: inheritance represented by an arrow pointing to the parent type, association depicted by a connecting line and a rhombus next to the field from which the association originates. As third connection the proposed diagram adds a

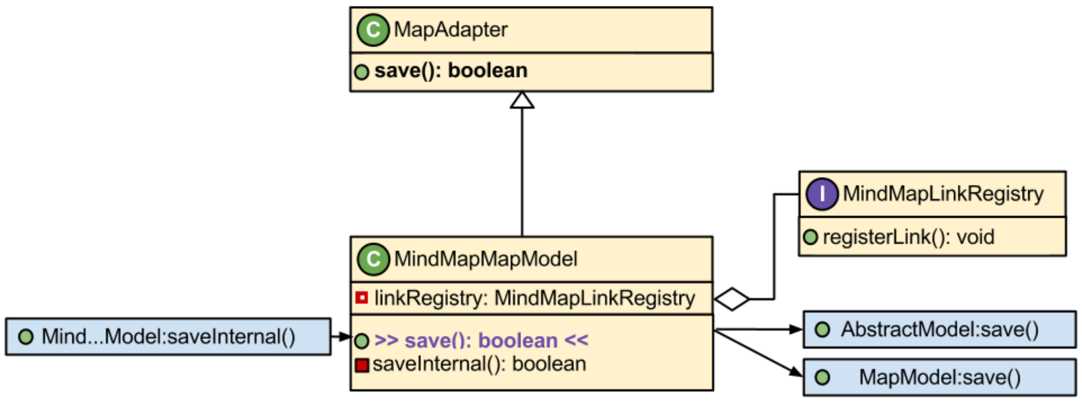


Figure 4.1: This example of the visualization for the proposed approach shows a chart similar to a UML class diagram with inheritance and composition. It also shows a call graph around the currently selected method (*save(): boolean*).

new element to the UML-class-diagram notation: a call represented by a small arrow from the calling method to the called method.

Also visible in figure 4.1 is an element categorized as important (*MapAdapter.save(): boolean*) and the call chain of the active method (*MindMapMapModel.save(): boolean*) with one method that calls it and two methods that are being called by (*MindMapMapModel.save(): boolean*).

When a developer clicks on a code element the corresponding code should be opened in a code editor, highlighting the selected element.

The interactive diagram can either be placed next to the code editor in an IDE or as a maximized window enabling a developer to focus on the model and only show the code of some elements in a smaller code window besides or half-way above the diagram.

4.3 Proactive Navigation Support

The results of the exploratory study showed that developers can complete their tasks faster when they use structured navigation aids to analyze unfamiliar code. Our approach provides the developer with three types of structural navigation:

1. **Type Hierarchies**. The structural dependency analyzer finds type hierarchies if the types are relevant both types are displayed in the diagram. A developer can use the interactivity of the diagram an click on code elements to show them in the source code editor.

2. **Associations**. When a relevant class possesses a field whose type is relevant, the diagram shows an association between the class and the type. A developer can either click on the visualized association (diamond symbol and connecting line) to show the field that is an instance of a relevant type or click directly on a code element of the two types.

3. **Call Chains**. Whenever a method gets selected, the method is highlighted in the diagram and methods that call the selected methods are displayed left of the method and methods that are being called by the active method are displayed right of the active method (see Figure 4.1). The call chain is click-able and refers the developer to the respective code.

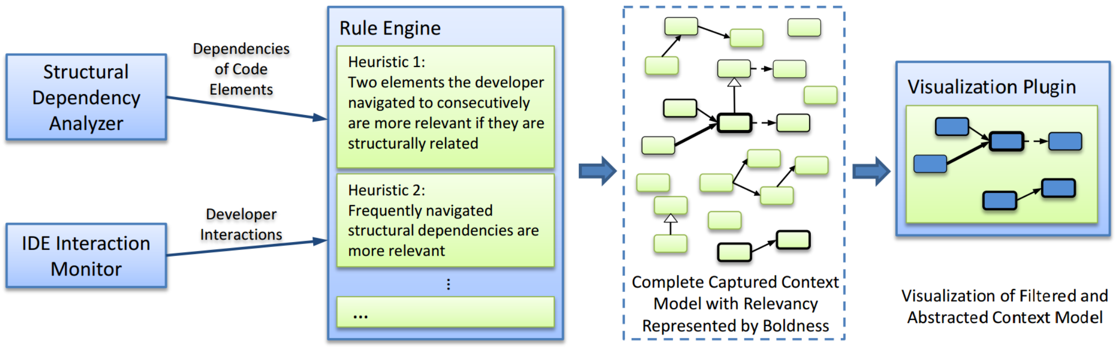
Next to the relevant elements that are likely to be visited again by the developer (as shown in the exploratory study) the methods in the call chain are not necessarily relevant and can help a developer browsing to unexplored code, thereby marking other elements as relevant. The elements that become relevant in this process are of course displayed in the diagram again which provides the developer with a comprehensive navigation support.

Chapter 5

Implementation

5.1 Components

The prototype is built in a loosely coupled design so that the main modules can operate within themselves. The communication between the modules is realized with an event model. To receive the developer's (the user of this eclipse plug-in) interaction over time trackers are registered to eclipse and send events to a central event dispatcher *EventBus* . All modules that need to react on events register themselves for the needed events to the *EventBus*. The module that evaluates the importance of the code elements is the rule engine which applies an internal list of heuristics after receiving events. The rule engine issues a *static dependency analyzer* which finds dependencies like associations and inheritance between code elements. These code elements are encapsulated in a *ContextModel* which notifies the *UI-Plugin* whenever elements of the context model change. To visualize the *ContextModel* two versions of graphic representations of the model were implemented as eclipse windows. (see Figure 5.1).



Structural Dependency Analyzer
Dependencies of Code Elements
IDE Interaction Monitor
Developer Interactions
Rule Engine
Heuristic 1: Two elements the developer navigated to consecutively are more relevant if they are structurally related
Heuristic 2: Frequently navigated structural dependencies are more relevant
⋮
...
Complete Captured Context Model with Relevancy Represented by Boldness
Visualization Plugin
Visualization of Filtered and Abstracted Context Model

Figure 5.1: Scheme of the prototype's implementation created in collaboration with and drawn by Thomas Fritz: A rule engine receives events from an IDE interaction monitor and structural code element dependencies. The rule engine then applies a set of heuristics to generate a model in which each code element is associated with a relevancy value. The elements with a relevancy value high enough are displayed in a visualization plug-in.

5.1.1 Interaction Monitor and Eclipse integration

To be able to listen to eclipse's events from the beginning when eclipse is started, we registered a start-up extension (IStartup interface) which registers several handlers to the relevant eclipse components. We implemented a list of events that were frequently used by the developers in the study and could be retrieved directly from the eclipse model:

- Opening a compilation unit
- Closing a compilation unit
- Bringing a compilation unit to the top (making it visible in Eclipse)
- Opening an editor (normally a Java editor)
- Closing an editor
- Bringing an editor to the top
- Selecting a Java field
- Selecting a Java method
- Selecting a compilation unit (Class or Interface)
- Navigating along a Java hierarchy
- Navigating along a Java Method call
- Pressing a key in an editor
- Scroll inside an editor

Since these events need to be gathered from different places of the eclipse model, a unified Interface (Tracker) was created and implementation of this interface can be added to a centralized register. Some of the events, however are issued multiple times when a developer only performs an action once. To get a better control over the events which are influencing the generation of the context model, an EventBus was implemented which is independent from Eclipse's event model. This EventBus allows to selectively forward only relevant events and eliminate duplicated events.

Events not only come directly from the eclipse's model but also from components of this prototype. For example when a developer clicks on a Java element which is displayed on the generated context model visualization then this element has a higher probability to be relevant. The selection event should therefore be forwarded to all components which need to know about it via the EventBus.

The events are ordered in a clear hierarchy so that listeners of the EventBus can selectively receive events and their sub-events. The event hierarchy is implemented with interfaces to allow events to be in multiple event categories at the same time.

The implementation of EventBus is similar and influenced by the one of Google's Guava libraries [gua] but has the ability to filter certain events. To issue an event the fireEvent method can simply be called like in this case where a Java Editor Selection Tracker notifies the selection of a Java method:

```
eventBus.fireEvent(new JavaMethodSelectedEvent(method));
```

Listing 5.1: Issuing an Event on EventBus

To receive Events a handler can be registered to the EventBus. In the example below the EventHandler listenes to all JavaElementSelectedEvents i.e. when a Java field, method, class or interface is selected in an editor. This example was taken from the visualized context model diagram, which selectively updates Java elements when they get selected:

```
eventBus.registerEventHandler(new EventHandler() {

   @Override
   public void onEvent(Event event) {
      JavaElement element = ((JavaElementSelectedEvent)event)
            .getJavaElement();
      updateElement(element);
   }

   @Override
   public Class<? extends Event> getEventType() {
      // listen to all JavaElementSelectedEvents
      return JavaElementSelectedEvent.class;
   }
});
```

Listing 5.2: Listening to Events forwarded by the EventBus

5.1.2 Abstraction of the Captured Java Element Model

For this prototype it must be possible to add a relevance property to all Java elements and to Java relations in order to filter non relevant elements in the diagram of the context model.

Instead of decorating the Java elements that are already modeled by the Eclipse model or the Java Reflection API, We chose to implement a new model from scratch where also Java relations are modeled as objects and therefore can have the relevance as property. This decision has the advantage to be portable to other systems (e.g. an IDE independent system or one for another IDE). Is is also a clear distinction of which elements are part of the context model and what elements are part of eclipse's syntax tree. It also has a simpler structure with only the elements needed for the context model and no decoration for the relevancy. Furthermore the model resembles the type hierarchy of the event model and can more easily be mapped. Other advantages are that it is more easily extensible and that it is not dependent on an other model which could change the model without notifying this prototype.

The model includes a Interface *JavaElement* for all Java elements which requires some methods to get or set the relevance for an element. Since these methods are the same for most *JavaElement*s they are implemented in an abstract class *AbstractJavaElement* which also insures the invariances for the relevance of an element by validating the values after each write access of the value. Direct ancestors of the *AbstractJavaElement* are *JavaClass*, *JavaField*, *JavaMethod* and *JavaRelation* which has three sub types: *JavaCallRelation*, *JavaComposition* and *JavaInheritance*.*JavaMethodAccessModifier* and *JavaFieldAccessModifier* are used to specify the access level of *JavaElement*s.

To make sure that *JavaElement*s can only be generated in the right context (e.g. there should only be a *JavaElement* if there is a corresponding element in the eclipse model) all constructors have the default access modifier which only allows them to be created inside the same package. Inside the same package there is a *JavaElementBuilder* which is the only class allowed to generate *JavaElement*s together with a corresponding class in the test folder *JavaMockElementBuilder*. The *JavaElementBuilder* can only create a new *JavaElement* out of a Java element of the eclipse model with help of the *JdtContextModelAdapter* which builds an interface between the context model and eclipse's JDT.

The central class in the prototype is *ContextModel* which stores all *JavaElement*s and offers several conveniance methods to query *JavaElement*s e.g. by their relevance. In order to keep the *ContextModel* in sync with the generated context model diagram the *JavaElement* implements the Observer design pattern [GHJV93].

5.1.3 Rule Engine

Because in the beginning of the development of the prototype it was not clear which heuristics would be most suitable to define the relevance of the code elements, we wanted to create a central point to define these heuristics. This point should also be flexible to allow optimizing the rules. For this purpose we defined a embedded domain specific language [FP06]. To make the language extensible and clear we described the language in an Java interface with inner interfaces in a way that the file looks similar to a definition in EBNF.

```java
public interface RulesDsl {

   TimeUnit every(int number);
   JavaElementAction onEvent(Class<? extends Event> event);

   interface TimeUnit {
      JavaElementSelector miliseconds();
      JavaElementSelector seconds();
      JavaElementSelector minutes();
   }

   interface JavaElementSelector {
      JavaElementAction withJavaElement(JavaElement element);
      JavaElementAction forAllJavaElements();
   }

   interface JavaElementAction {
      void increaseRelevancyBy(double number);
      void setActive();
      void findStructuralDependencies();
      void log(String logMessage);
      JavaElementAction forAllJavaElements();
      JavaElementAction ifSingleton();
   }
}
```

Listing 5.3: Definition of the DSL as nested Java Interfaces

```
(* int, double and String are defined in the Java Language *)
(* Event and JavaElement are omitted here and can be any
class extending Event and JavaElement, respectively *)

Rule                 = 'every(' int ').' TimeUnit |
                       'onEvent(' Event '.class).' JavaElementAction;

TimeUnit             = 'miliseconds()' | 'seconds()' | 'minutes()';

JavaElementSelector  = 'withJavaElement(' JavaElement ').' |
                       'forAllJavaElements()';

JavaElementAction    = 'increaseRelevancyBy(' double ');' |
                       'setActive();' |
                       'findStructuralDependencies();' |
                       'log(' String ');' |
                       'forAllJavaElements().' JavaElementAction |
                       'ifSingleton().' JavaElementAction;
```

Listing 5.4: The same definition in EBNF

The ability to configure global settings that have an influence on the relevancy setting is implemented as static methods in a separate *Relevance* class and can be added to the rule engine with a static import.

The *Relevance* class adds the properties to the logic of the prototype. The *relevancyThreshold* defines whether an element is relevant (all elements which have a relevance value over *relevancyThreshold* are candidates to be displayed on the diagram). The *minimalRelevancy* makes sure code elements do not get a very low relevance value. If a code element would have such a value it would take too long to make it relevant again and display it on the visualized context model. The third value the *Relevance* class adds to the model is an *importanceThreashold*. All elements with a relevance higher that *importanceThreashold* are displayed more prominently in the diagram.

With this rule engine the degree-of-interest model as defined in [KM05] can be easily implemented. The degree-of-interest model of Mylar (the predecessor of Eclipse Mylyn [myl]) defines three rules:

- Whenever a code element is selected the degree-of-interest for this element increases by one unit and all other elements in the model decrease their degree-of-interest by 0.1.
- Whenever a key stroke is issued, the affected code element increases the degree-of-interest by 0.1.
- Only display elements with a degree-of-interest over -10.

In the DSL of the rule engine of this prototype the code would look like that:

```
import static ch.uzh.ifi.seal.contextmodels...Relevance.setRelevancyThreshold;
//...
setRelevancyThreshold(-10);
onEvent(JavaElementSelectedEvent.class).increaseRelevancyBy(1.1);
onEvent(JavaElementSelectedEvent.class).forAllJavaElements()
      .increaseRelevancyBy( -0.1);
onEvent(JavaElementKeyPressEvent.class).increaseRelevancyBy(0.1);
```

Listing 5.5: Mylar's degree-of-interest model implemented in the DSL

The Mylar degree-of-interest model was used as the starting point to generate heuristics for this prototype.

Heuristics

To come up with a set of rules we started with the rules that were implemented in [KM05]. We then used this rules to generate a set of relevant elements by simulating the developer interaction from a transcript that was recorded during the study. We incrementally altered the rules to get better results in terms of the number of elements that are displayed and what elements were categorized as relevant. To reverence of relevant elements, we took the developer models drawn by the same subject of which we used the transcript to simulate the interaction steps. We also altered the initial set of rules based on the insights gathered from the exploitative study as well as we used our subjective impression of the outcome of the simulations. After altering the heuristics, the size of the diagram was about the average size of the developer models from the study (about 4.1 classes). As a second test, all navigation steps of the developers were ran through and the end result was again compared to the developer models. After these tests and a second iteration of altering the heuristics we came up the following set of heuristics:

- Relevancy Threshold: 0 (only elements with relevance over 0 are being displayed in the diagram).
- Importance Threshold: 2 (elements with a relevance over 2 are being highlighted)
- Minimal Relevancy: -0.9 (no element can get a relevancy under -0.9)
- Increase Relevancy of an Element by 1 when the Element is Selected in any Eclipse Window or Editor (including the diagram).
- Decrease Relevancy of all other elements by 0.1.
- Set Elements Active when they are Selected.
- Whenever an Element Becomes Relevant, find Structural Dependencies (Composition, Inheritance and Method Calls) of that Element.
- Increase Relevancy by 0.1 when a Connection to an other Relevant Element is Found.

5.1.4 Structural Dependency Analyzer

The static code analysis is handled by a syntax tree analyzer which uses eclipse's Java Development Tools [jdt] to retrieve the relevant elements from eclipse's syntax tree. The basic functionality is analyzing type hierarchies, type compositions and method call hierarchies. While the class relations are used to display the structure, the call hierarchies should help the user of the prototype to see methods which might be relevant for the current task. The AST Analyzer issues an event whenever a new relation is found which can then be handled by the rule engine and finally influence the relevance of elements and relations.

In future work the AST analyzer could be used to analyze more complex structures like design patterns which could then be presented in a diagram and give the developer a deeper insight into the design of the source code.

In the current prototype there is an implementation of a Singleton [GHJV93] design pattern detector. This implementation is thought more as a proof-of-concept than a useful feature. It shows that the design of the prototype is easily extensible. To access the feature a corresponding method which calls the singleton detector was added to the rule engine.

5.2 Visualization

5.2.1 Initial Design Requirements

Not all information to develop the prototype was available when we started developing it. While some design considerations were clear from the beginning, the exact heuristics that have to be applied to determine the relevant elements and the appearance and functionality of the diagram were not clear. In this paragraph a list of design considerations which should not be altered during the development process is listed:

Integration into Eclipse. If the visualized context model should be used by developers in their every-day work it should become a well connected part of Eclipse. The same icons that Eclipse uses should also be used so that users see visually that the elements of the diagram are linked to Eclipse's Java Editor.

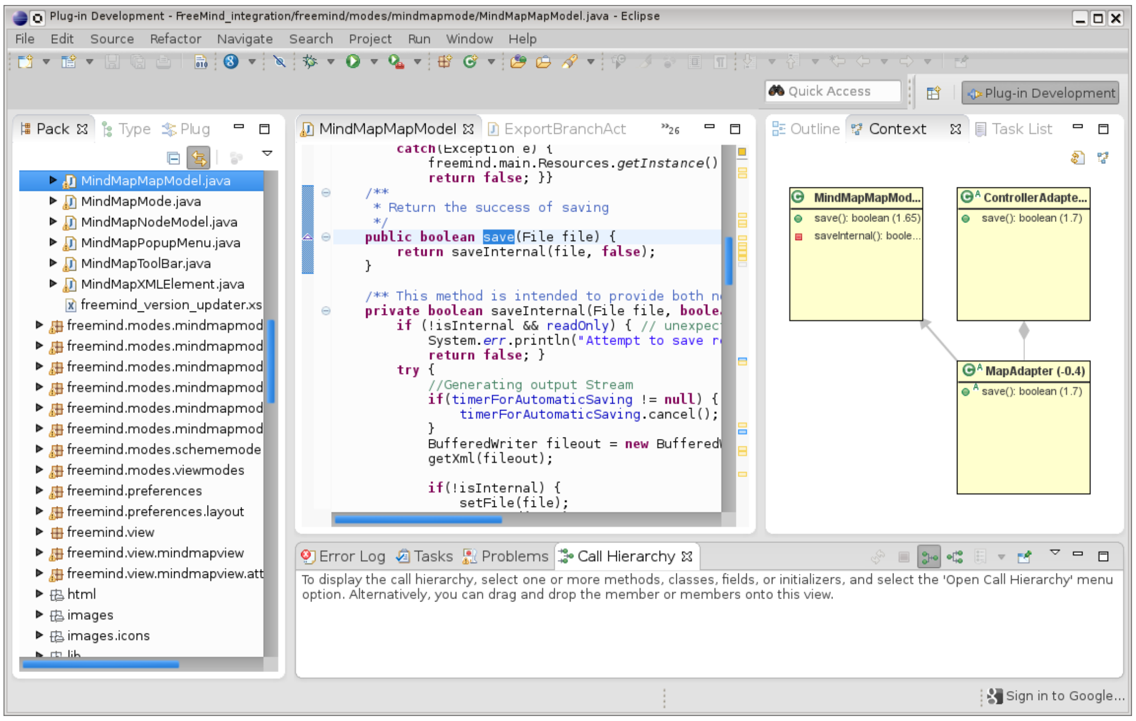
Interactivity. The developer should have the possibility to click on code elements. We already know that developers often revisit code but we are still not so sure about what information they exactly need in a revisit. Probably the structure of the diagram and the name of the code elements is not enough information. When a developer clicks on a code element this element should be opened in the appropriate editor (e.g. a click on a Java Method should open the Java Editor and scroll to the element inside the class).

Intuitiveness. Often users stop or do not even start using a software if it is not intuitive from the beginning. A visualization can be very powerful and show all relevant data within a small area but it might have to be learned first. In software development there exist a lot of well known notations like Unified Modeling Language (UML) [BJR00]. Even if the visualization only resembles a well known notation and does not implement it in all details it will be easy for developers to understand it without a prior learning phase.

Responsiveness. The prototype should be designed to help developers to compete tasks faster. The plug-in has to take a lot of considerations (heuristics) into account when determining the relevant code elements. Even tough there is a lot of data to be processed the visualization should work in real time and the users should not have to wait after each navigation step for yet another Eclipse plug-in.

To us, the most basic idea to develop such a prototype was to to visualize the relevant code elements in a UML-like [BJR00] diagram and show this graphic next to the Java Editor in Eclipse (see Figure reffig:prototype1). The diagram would be updated every time a developer interacts with Eclipse in a way that the relevance of a code element changes according to simple heuristics.

To start with, we used the same heuristics as described in [KM05] for the Mylyn project (see Page 23).



```
        catch(Exception e) {
            freemind.main.Resources.getInstance()
            return false; }}
    /**
     * Return the success of saving
     */
    public boolean save(File file) {
        return saveInternal(file, false);
    }

    /** This method is intended to provide both n
    private boolean saveInternal(File file, boole
        if (!isInternal && readOnly) { // unexpec
            System.err.println("Attempt to save r
            return false; }
        try {
            //Generating output Stream
            if(timerForAutomaticSaving != null) {
                timerForAutomaticSaving.cancel();
            }
            BufferedWriter fileout = new BufferedW
            getXml(fileout);

            if(!isInternal) {
                setFile(file);
```

To display the call hierarchy, select one or more methods, classes, fields, or initializers, and select the 'Open Call Hierarchy' menu option. Alternatively, you can drag and drop the member or members onto this view.

Figure 5.2: The Prototype showing three classes with methods and structural dependencies.

Technical Background

The prototype was implemented using the Zest toolkit [zes] which is part of the Eclipse's Graphical Editing Framework [gef]. Zest includes a set of visualization components which can be used in Eclipse. It especially includes the ability to draw directed and undirected graphs and several algorithms to layout the nodes of the graph. Some limitations of Zest came apparent during the development of the prototype: the included layout algorithms are either changing the the whole layout after each small alteration of the model (spring layout), use too much space between nodes are not suitable for a UML diagram (e.g. Tree Layout or Radial Layout). Another limitation of Zest is that it was hard to alter the decoration of the edges of the graph. for our UML Diagram we wanted to have at least composition and inheritance implemented into the diagram which would require an arrow for inheritance and a diamond for composition. Since the relevant code elements in Zest were set private in an abstract class we had to use Java's reflection API to access these fields. Finally it was impossible for us to resize nodes of the graph although there is functionality built into Zest the elements disappeared completely when custom graphics were used for nodes and the option that they should keep their original size was enabled.

Because Zest is built upon Draw2D [dra], which is also part of the Graphical Editing Framework is was easy to draw very detailed graphics for the classes using Eclipse's icons.

Functionality

The visualization of the prototype shows all relevant types (which have a relevancy over the relevanceThreshold field in the ContextModel class). Then the types get connected with each other if there was a composition or inheritance found. If both a inheritance and a compositional connection was found between the same two classes, only the inheritance is shown. The inheritance is analyzed with Eclipse's built in Type Hierarchy functionality it is displayed with an arrow between the two types pointing to the parent type. Compositions are found by analyzing the fields of the relevant classes. If such a field is of the same type as displayed relevant type then this is categorized as a composition and displayed with a connecting line between the types and a diamond symbol on the connection next to the class with the field. If several of these connections are found between two types, only one is shown. By clicking on a connection with an arrow, the field is selected in Eclipse's Java Editor. Inside the type boxes in the diagram the relevant fields and methsods are shown each with a symbol (the same symbol as used in other Eclipse plug-ins) indicating the properties of the members (access level, abstractness and so on). Clicking on a method or field opens the corresponding source code location in the Java Editor of Eclipse.

Proactive Navigation Support

Whenever a method is selected, either in the source code or on the diagram, this method is being highlighted and two blue boxes pop up which show an ordered list of methods that call the selected method and in the other box methods that are being called by the selected method. The methods are ordered in a why that the one more likely to be selected next is on top. This is not the relevancy used by the rest of the diagram, because these navigation to methods could be done over more than one step i.e. the developer could reach a more relevant method by first navigating to one method which will then show the developer a more relevant method. To calculate this relevancy the *ASTAnalyzer* calculates all possible paths over three steps, sums up the relevancy values of all methods in the path, the it builds the average over all summed relevancies that have the same method as origin. This average is then used to sort the methods inside the caller and callee boxes.

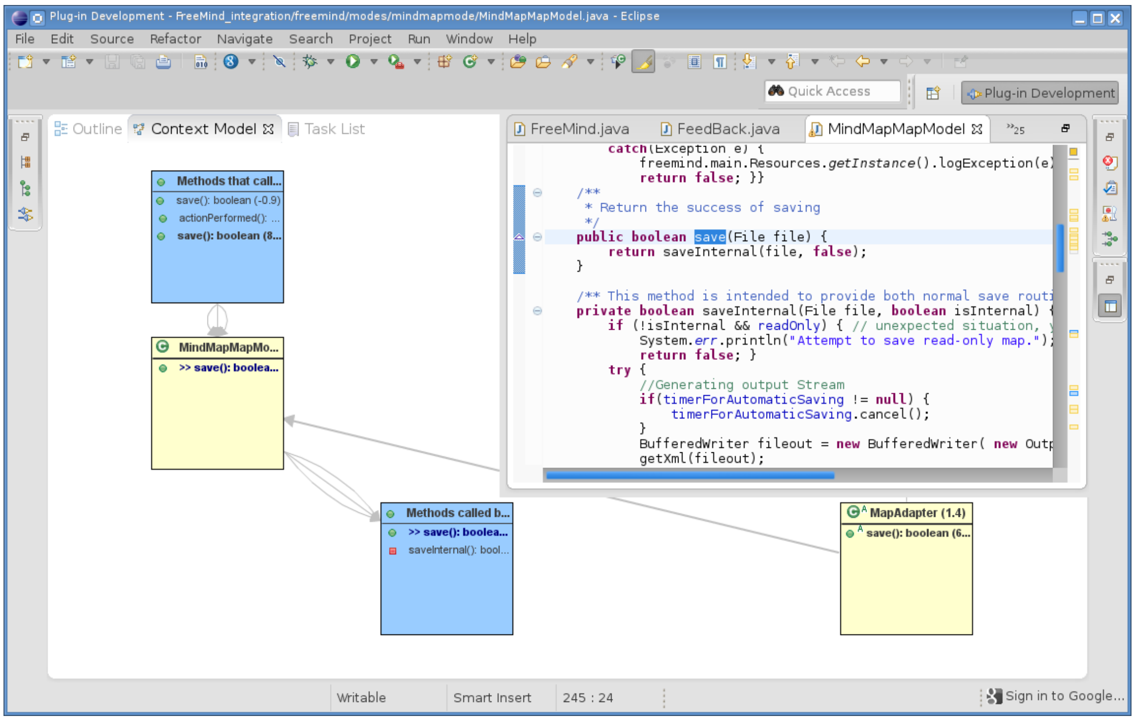
Alternative Usage

While the plug-in was initially indented to be displayed next to the Java Editor in Eclipse, the idea came up that with a sophisticated visualized context model, this model could be located in the center of the development environment (see Figure 5.3). This window reflects the estimated model that the developer has in his mind and should therefore give the developer a good overview of the current task. When the developer clicks on an element in the model, the code is shown next the diagram and can be edited.

This way the plug-in could in many cases replace the Package Explorer and if the model would be composed well enough be the most important point of navigation. This however would require a much more detailed view with more elements but could certainly be matter of future work.

Shortcomings of the Prototype

While working with the first iteration of the prototype it was noticeable that the layout of the diagram changed substantially when the developer navigates to a different class. Although in most cases it seams easy to associate the diagram with the code again after such a change in the layout, the developer needs time to comprehend the diagram and gets distracted from the actual task.



```
        catch(Exception e) {
            freemind.main.Resources.getInstance().logException(e
            return false; }}
    /**
     * Return the success of saving
     */
    public boolean save(File file) {
        return saveInternal(file, false);
    }

    /** This method is intended to provide both normal save routi
    private boolean saveInternal(File file, boolean isInternal) {
        if (!isInternal && readOnly) { // unexpected situation, y
            System.err.println("Attempt to save read-only map.");
            return false; }
        try {
            //Generating output Stream
            if(timerForAutomaticSaving != null) {
                timerForAutomaticSaving.cancel();
            }
            BufferedWriter fileout = new BufferedWriter( new Outp
            getXml(fileout);
```

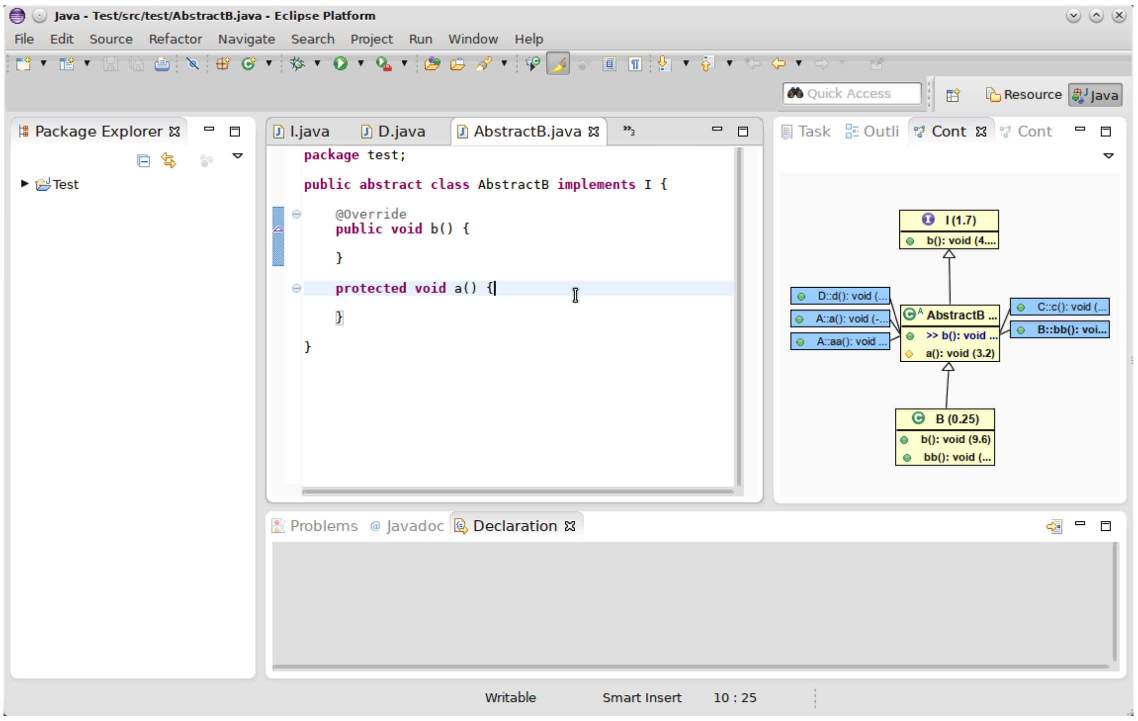
Figure 5.3: The Prototype showing the Visualized Model in Full Screen and the Associated Code in the Corner

Also since with the chosen framework all boxes in the diagram have the same size, precious space gets lost between the elements and not all names inside the boxes are readable anymore. While this shortcoming is not that relevant in full-screen mode or with a high screen resolution, when the diagram is displayed next to the Java Editor with a lot of relevant elements, the problem becomes more present.

5.2.2 Improvements to the Prototype

With a second iteration of the prototype, we wanted to focus on the original usage again - displaying the diagram next to the Java Editor as a visual representation of the context model and as a navigation help which also provides proactive navigation support. But the shortcomings of the first iteration of the prototype should be resolved as well as possible. Since the layout algorithm of the Zest toolkit and the strictness of it (e.g. that the decoration of a connection can hardly be changed) were the cause of most of the limitations, we used Draw2d directly this time this for instance allowed the classes to be drawn in their original size so that not all classes have the same size set by a layout algorithm. The heuristics are the same as for the first iteration of the prototype, only the visual representation changed for the second iteration of the prototype.

To make the diagram more static, a fixed layout scheme was chosen. The whole window is divided into 9 sectors (three columns and three rows). The active class (e.g. the class that is selected in the Java Editor or the class that was clicked on in the diagram) always is displayed in the center of the screen. If a method is active the calling methods are displayed left of the active class and the methods that are being called by the active method are displayed right of the active class. In a next step, relevant super types of the active class are displayed in the top row of the



```
package test;

public abstract class AbstractB implements I {

    @Override
    public void b() {

    }

    protected void a() {

    }

}
```

Figure 5.4: The second iteration of the prototype showing detected inheritance between classes and a call chain of the currently selected element.

window (starting with the middle sector in the top row). Relevant subtypes are then displayed below the active class (again starting with the middle sector of the bottom row). The remaining sectors are first filled with relevant classes the have a connection (e.g. a composition) to the active class and then with relevant classes that have no connection the the active class. In Figure 5.4 5 of the 9 sectors are shown (active class in the middle, interface of the active class on top, subtype on the bottom, callers and callees on each side of the active class). Connections are always drawn to the active class. Other connections are only drawn, if they don't cross another class. The callers and callees are now separated and drawn in a box for each method.

Shortcomings of the Second Iteration of the Prototype

In comparison to the first iteration of the prototype, the prototype with this changes is highly focused on the active element. Relevant classes that are not connected to the active class but are connected to another relevant classes might not be connected in the diagram. That puts the focus away from the composition of the model but makes the diagram more static and less jumpy. In the first iteration of the prototype, all callers and all callees were grouped in one box, even if they are not in the same class. The second iteration of the prototype separates all these methods into single boxes even if they are in the same class. That allows the methods to be sorted according to their relevance but again puts the focus away from the structure of the model. Also if a method is already displayed in another relevant class the method can appear twice in the diagram without pointing this duplication out to the developer.

5.3 Technical Aids for the Implementation

5.3.1 Evaluation Support

Since the whole system is only triggered through events, a developer's navigation through the code can be simulated by sending a series of events to the *EventBus*. To find the code elements that are needed for the events, a utility method was added for each type of code element (e.g. *class(String fullyQulifiedClassName)* or *method(String fullyQulifiedClassName, String methodName, String... argumentTypes)*). Since only the actions in the transcriptions are needed that have an influence on the diagram, no additional Events had to be implemented for the evaluation support. An example for a formalized transcript could look like that:

```
public class Transcript9 extends AbstractTranscriptRunner {

   @Override
   public void runTranscript() {
      fire(new JavaClassSelectedEvent(
            clazz("freemind.modes.ControllerAdapter")));
      fire(new JavaMethodSelectedEvent(
            method("freemind.modes.ControllerAdapter", "save")));
      fire(new JavaMethodSelectedEvent(
            method("freemind.modes.MapAdapter", "save")));
      fire(new JavaMethodSelectedEvent(
            method("freemind.modes.mindmapmode.MindMapMapModel", "save")));
      // ...

   }
}
```

Listing 5.6: Example for simulated developer code navigation

To be able to detect re-visitations we added a *navigation history* which stores all elements that are navigated through in a simulation. The *navigation history* can be extended with triggers that listen to specific events like the code event of the current developer interaction was previously visited 3 times.

We also added support to automatically calculate precision and recall between the generated model and a random model that can be added to the evaluation support module. Since we implemented the module to later compare the models drawn by the developers during the study with the generated model, the module was called *developer model*.

To allow us to not only see the model at the end when the whole developer interactions passed, we added functionality that stops after a certain amount of steps and waits until the user continues with the evaluation. By default the number of executed steps was is to 10. This can also help when simulating very long transcripts. The structural dependency analyzer traverses all dependencies of a code element which is very memory consuming. In a large transcript, when executing all interactions after each other without a break, the garbage collector fails to free the memory.

5.3.2 Testing

To separate test code from production code we used maven to structure the project into three modules. A parent module which keeps the common settings and two child modules, one fore production code and one for test code. That way we could also separate the dependencies (e.g. packages like JUnit [jun] should not be part of production code). The use of maven also made it easier to continuously test the integration.

Chapter 6

Evaluation

The purpose of the presented eclipse plug-in is to help developers understand the code relative to a given task. In the exploratory study, we showed that developers need to revisit code often to understand the relations between code elements. To measure how the plug-in could reduce such revisits, we enabled our plug-in and replayed the user interactions with the IDE that are recorded from the study. Our system detects if a code element was visible in the diagram each time when the developers revisited that code element.
We also analyzed the diagrams generated during the replayed interactions. In the study the subjects provided us with the model about relevant code elements in their mind(see section 3.1.2 - Developer Models). We compared the extracted elements of the developer models and compared them with the elements classified as relevant by the plug-in after each developer interaction that had an impact on the plung-in's internal model.
Building an unnecessarily large model would not help the developers much. Therefore we also analyzed the size of the visualized model. On the one hand it should have a similar size to the developer model, as it should carry the same amount of information the developers had in mind. On the other hand it should not exceed the limited space of the window and thus make the elements too small to read.

6.1 Method

To simulate the same interaction that the developers of the exploratory study did, we automatically extracted a list of events that could be received by the plug-in as described in Section 5.3.1. Since the transcripts did not include the fully qualified class names and several class names appear multiple times in the code, the list of events had to be manually extended by the fully qualified class names. The transcripts contain a lot of steps that are not (yet) handled by the prototype like scanning or debugging steps. Therefore only steps that have an influence on the diagram were simulated for this evaluation. The automatic calculations in the plug-in include the precision and recall values between the elements of the developer model (acting as relevant elements) and the code elements that were classified as relevant by the plug-in (acting as retrieved elements). For that purpose the elements of the developer model had to be added to the plug-in. To determine the number of revisits that were shown in the diagram at the time of revisiting the element, a navigation model was added to the plug-in, which stores a history of all navigation elements and which is completely uncoupled with the context model (so it does not have an influence on it). That way all revisits can be determined even if the element is not yet in the context model of the plug-in. Then the element of revisit looked up in the context model. If it is categorized as relevant in the context model (and thus shown in the diagram) it is added to the number of revisits that were shown in the diagram during the re-visitation.

6.2 Revisits

From the in average 54.4 steps that were simulated for that evaluation, 19.8 were revisits. During the simulation, all revisited elements were visible in the diagram before they were revisited (see table 6.1). To see if the developers would not need to revisits the elements because they were visible in the diagram, an extended study with new subjects would have to be performed. To get comparable results from such a study we would need a statistically significant number of subjects and the previous study also would have to be extended to have enough developers. The presented numbers only show the maximal number of revisits that could possibly be eliminated. From personal experience, we would assume that only a fraction of the revisits could effectively be prevented because only the structure of the elements is shown in the diagram and not the code. Since the diagram was designed so it is interactively usable, the revisits could be executed in a more structured way and developers with a distinct visual memory could probably keep the structure better in mind.

Project	*ID*	*Job*	*Years*	*Time*	*Dev. Model*		*Diagram*		*Nav. Steps*	*Simulated Navigation Steps*		
			Pr.Exp.		*Cl*	*Me*	*Avg. Cl*	*Avg. Me*	*All*	*All*	*Revisits*	*D*
Freemind	**F1**	pro	10	39.7min	4	0	5.12	5.8	116	93	26	26
	F2	pro	11	22.0min	4	0	5.8	6.27	106	44	4	4
	F3	grad	8	59.7min	5	2	4.72	3.51	341	61	20	20
	F4	grad	9	70.9min	4	3	5.81	2.39	177	120	56	56
JPass	**J1**	pro	12	8.6min	4	5	4	1.75	67	8	0	0
	J2	pro	12	64.5min	4	0	5.25	1.3	408	20	7	7
	J3	pro	11	62.0min	5	7	3.51	1.25	140	55	26	26
Rachota	**R1**	fac	15	53.8min	6	0	2.63	4.19	349	29	10	10
	R2	grad	11	100.6min	4	6	3.13	0	553	52	16	16
	R3	pro	15	36.6min	6	0	3.61	4.47	326	62	33	33
Average					4.6	2.3	4.36	3.09	281.5	54.4	19.8	19.8

Table 6.1: This table presents a summary of the results collected from the evaluation of the prototype, compared with the results of the exploratory study ($pro = professional$, $grad = graduatestudent$, $fac = faculty$, $Cl = classes$, $Me = methods$, $D = RevisitsVisibleInDiagram$).

6.3 Relevancy

The models drawn by the subjects of the exploratory study do not follow any common structure and have different notations. Some show which classes are used in which order, others make use of a UML-like notation. One thing that all drawings have in common are classes. To be able to compare the diagrams in a unified way we only used classes in a unordered set and compared it to the classes categorized as relevant in the plug-in. We also only used the transcripts of those developers from the study that succeeded with their task for the evaluation so that the data can be used to later improve the prototype. In our result, 2 out of 12 subjects are eliminated. A unsuccessful developer could in some cases drive the tool in a direction that it would not help developers but make them fail when following the recommendations. Surprisingly, when rerunning the transcripts, the developers that worked with the Freemind-task already had a good precision and recall values after a few steps (See Figure 6.1). For this task three out of four developers found the relevant elements started with a higher precision after a few initial steps and then they ended the task. The precision usually declined when the developers started to edit code. Then the automated eclipse tools helped the developers navigate to further code elements. For example when a developer changes the signature of an inherited method then eclipse's quick-fix functionality automatically leads the developer to the parent class, which might not be relevant to understand the context of the task but is needed in order to complete the task. In most simulated sessions

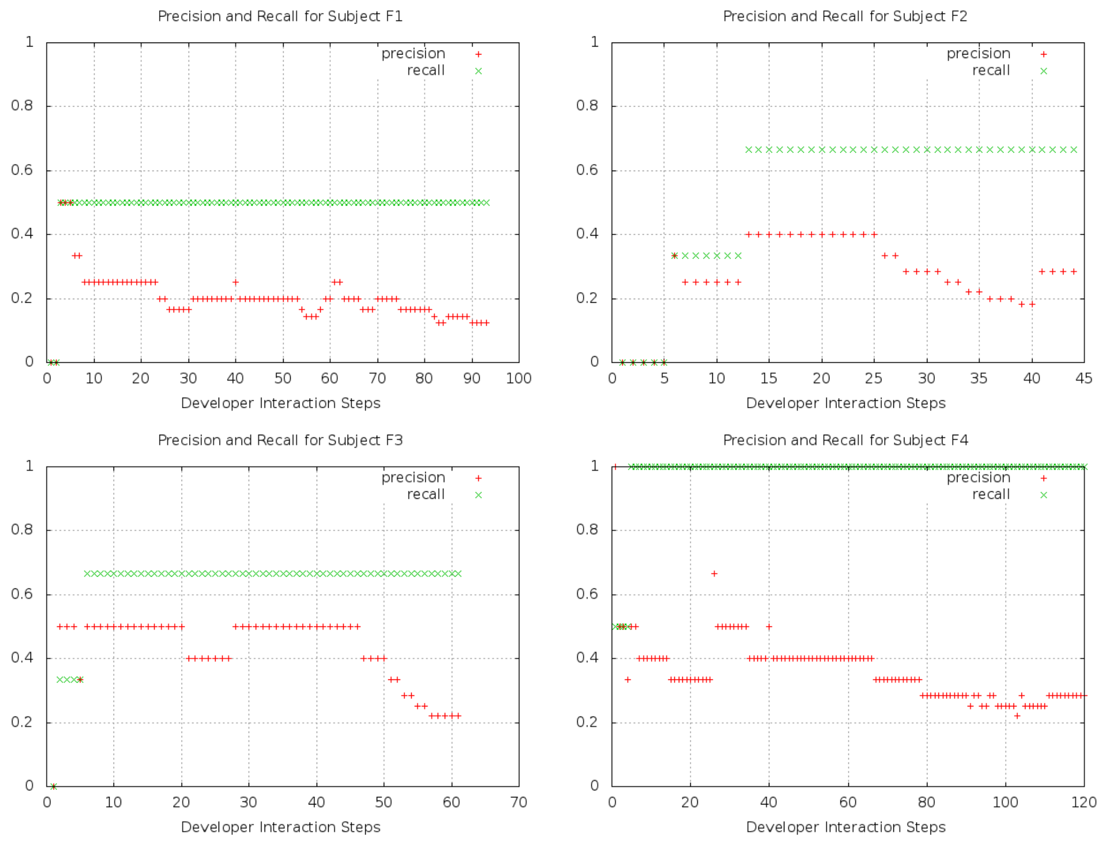


Figure 6.1: Precision and Recall for the Subjects Working on the Freemind-Task (Relevant Elements: Classes of Developer Models, Retrieved Elements: Classes Categorized as Relevant by the Plug-in)

the recall values are significantly higher than the precision values. These values show that some of the classes that were relevant were not shown in the diagram. The model of the plug-in was designed in a way that there should not be too many classes so that they can fit to the screen. In average of all simulated runs, the diagrams showed 4.36 classes, which is about the same size of the developer implicit models (4.6 classes in average).
The precision and recall values for the Jpass and Rachota tasks (Figures 6.2 and 6.3) are close to our expectation. These values start lower and raise until the last navigation steps. Also the values are better than the ones of the Freemind task, except the one of Subject J2, which is surprisingly low. The developer model of subject J2 looks similar to the ones of the other developers, but during the navigation, the subject spent most time going through each class folding the methods in Eclipse's JavaEditor only leaving the ones open that seemed relevant the the Subject. Since we evaluated the navigation of all developers the same way we did not especially extract the methods that seemed relevant to subject J2. Also the folding and unfolding actions were not captured with the eclipse plug-in, what explains the low number of steps with more than one and

half hour work on the task. After a long time folding and unfolding methods (one hour and 15 minutes), the subject started to work more similar as the others which can also be seen in the precision and recall diagram in step 13, where both precision and recall raised slightly. Subject J1

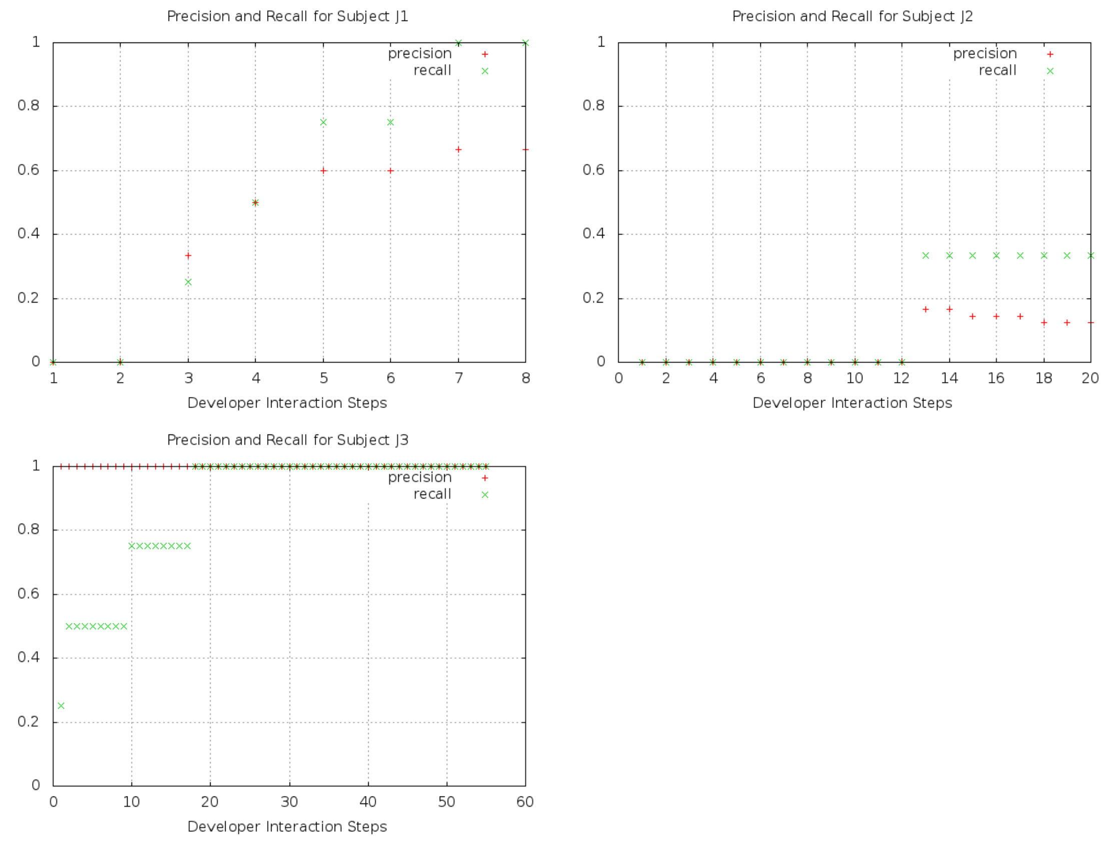


Figure 6.2: Precision and Recall for the Subjects Working on the JPass-Task (Relevant Elements: Classes of Developer Models, Retrieved Elements: Classes Categorized as Relevant by the Plug-in)

only performed 8 steps that were captured by the eclipse plug-in. This developer was extremely successful and finished the task within 8.6 minutes. As we can see in the precision and recall diagram (Figure 6.2), the developer navigated to a precision value of 0.67 and a recall value of 1.0 within only 8 captured steps. Each step increased both precision and recall (except the last one that had a slightly lower precision value). None of the captured steps was a revisit. Presumably, there is little help that the proposed tool can provide to developers that are skilled like Subject J2 in an easy task like the JPass task.
Subject J3 came to the highest possible values for precision and recall within less than 20 steps, staying there until the task was fixed. That result shows that the developer used most of the navigation steps in the same classes that the developer drew in the developer model.

The diagrams for the Jpass and Rachota tasks (Figures 6.2 and 6.3) also do not show the same decline of precision and recall in the end of the tasks. That is possible because the changes of the code lie directly in the classes drawn by the developers in the developer models.

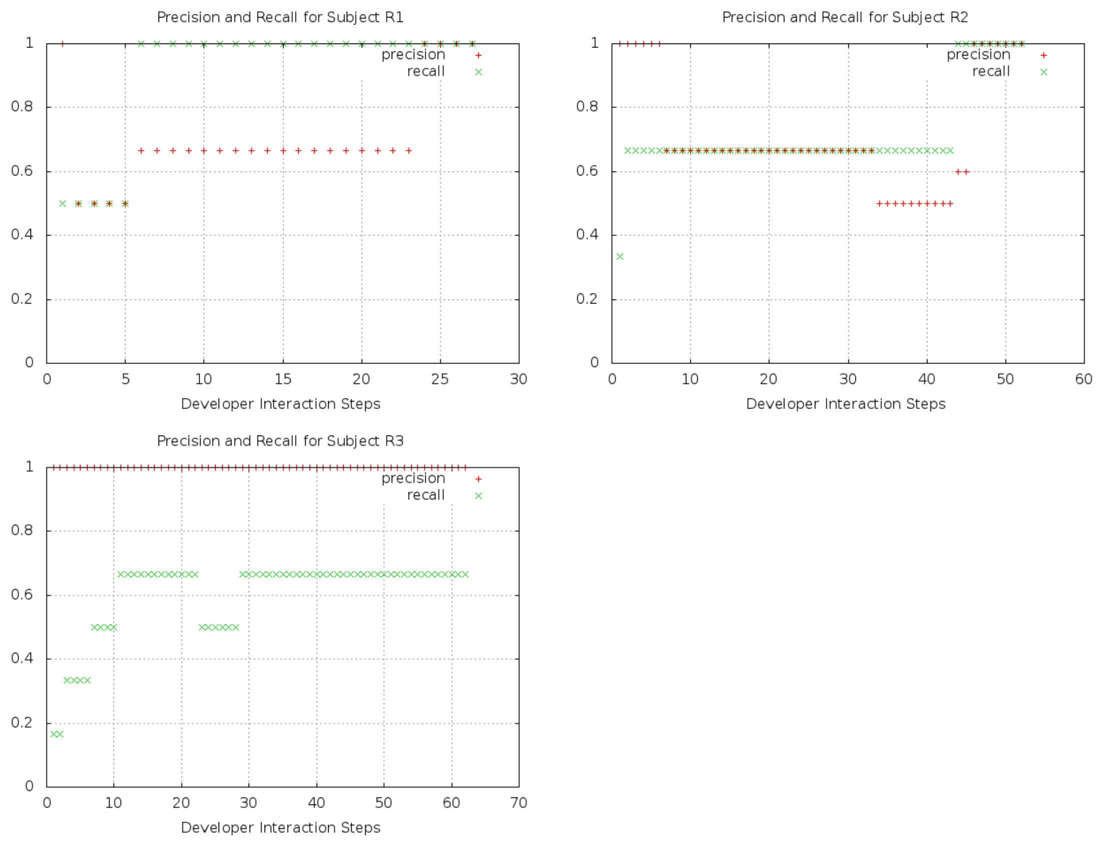


Figure 6.3: Precision and Recall for the Subjects Working on the Rachota-Task (Relevant Elements: Classes of Developer Models, Retrieved Elements: Classes Categorized as Relevant by the Plug-in)

6.4 Visualization and Interactivity

We compared the generated visualization with the models drawn by the developers in the exploratory study to see if the model fits into the available space in the Eclipse IDE.
The diagrams drawn by the study subjects were all very different but mostly they resembled a UML-like diagram with classes and sometimes also methods. The diagrams did not discriminate between a lot of different relationship types but usually used a generalized form of inheritance and association. The visualizations of both prototypes also only use these two forms of relationships.
In average, 4.1 classes were drawn in each developer model (4.6 classes in the diagrams of the successful developers). The relevant elements in the generated context model of the Eclipse plug-in contains 4.36 classes in average. 7 out of the 12 developer models contained methods, in average 3.86 (only 5 of the models of the successful developers contained methods). The Eclipse plug-in contained 3.09 methods in average during the simulated developer interactions. The generated visualization and the developers' drawn model have similar numbers of classes and methods. These classes and methods are easily placeable in a small window beside the code in the Eclipse IDE.

To evaluate the usefulness and effectiveness of our approach, in particular with respect to its visualization and interactivity, a user study with developers using our prototype would be required. This will be further investigated in future work.

Chapter 7

Discussion

7.1 Threats to Validity

The exploratory study was conducted with 12 participants, six from industry and six from academia. From the 12 developers 10 completed their task successfully. The study subjects were assigned to three tasks resulting in three or four successful developers per task. This number of developers does not allow us to make statistically significant statements about developers in general and having three tasks does not allow us to make statistically significant assertions about tasks in general or bug fixing tasks in particular. The number of developers and tasks however allows us to get an exemplary insight about how bug fixing tasks are handled. Together with personal experience it also allowed us to find participants that work in a uncommon way, as subject J2 who used the code folding functionality in the source code to mark the relevant methods (see Section 6.3).

The prototype was only evaluated using the previously gathered information from the exploratory student. It was never in a explicitly demonstrated to developers and it was never used to perform a task like the ones investigated in the study. Also the performance of the prototype was only tested on the authors computers and only with the tasks from the study. From this data we can not asses if developers are willing to use the prototype. This would depend on several factors like if the visualization is intuitive enough, if it runs with adequate performance and does not slow down the developer and if the developers could associate the code elements in the diagram with the actual code elements, to just name a few factors. These factors and more would have to be tested in an extended study.

7.2 Future Work

Improve Heuristics

To have a better assessment of the relevant and important code elements obviously, the heuristics could be improved. A larger study with more developers that point out which elements are important for them could be used as base data. Moreover with the automatic recordings of the eclipse interaction the correlation between each type of interaction on a element to the relevance (given by the developer models from the study) could be calculated. Out of the correlations, the heuristics can easily be generated.

Share Navigation History to get Better Results

Since on large projects, where it is particularly difficult to get an overview of the relevant elements usually multiple developers work on the same files. The interaction of all developers could be recorded and uploaded to a central server. The server could then look for similar tasks that have already been completed and could show the relevant elements from the similar task of an other developer or of the same developer from a past task.

Incremental Exploration

In Relo [SKM06] developers are able to explore the code in an UML like diagram by clicking on buttons which are embedded in the diagram and lead the developer to more information. Such an approach could also be implemented into this work. Irrelevant code elements are already filtered out but they might be classified as irrelevant because the developer never navigated there. If the developer would not only have the possibility to explore new code in the source code editor but also in the diagram, it would be possible to see code not recognized as relevant in the diagram without first being explored in the source code editor.

Showing the use of Common Structures

To make code better understandable, sometimes developers name code elements after best practices like design patterns (see [GHJV93]) to help other developers understand the structure easier. The prototype could use automated design pattern recognition (e.g. as presented by as a research team of the IBM Software Solutions Division in Toronto Canada [BFVY96]). The recognized design patterns could be visualized in the diagram. A singleton class, for example could be annotated with an icon indicating the pattern or a factory pattern could be indicated with an additional *"creates"*-relation between the factory and the types that it generates.

7.2.1 Considerations for the Layout of a Future Prototype

Visualization and Interactivity

Both previous prototype have advantages. The first prototype layouts the model, focusing on the structure (inheritance and composition) and does a better job with that. The second prototype has a more static layout where all the elements in the diagram have their fixed location but the connections between the elements are not always shown. Also both prototypes do not scale very well when the window is resized. The first prototype does a better job here by enlarging the classes but neither prototype shows more elements when the window has more space.

For a future prototype we propose a grid layout similar to the second prototype which should (additionally to the second prototype) be scalable so it shows more elements when the window has more space. If the window size is larger and more elements can be placed in the diagram, the more relevant elements should be located in the middle of the window around the active class. When more classes are shown in the window it is also possible to arrange the classes in a way that more connection can be shown without crossing classes. To mark the importance of classes, the size of the classes can be adjusted to the relevance (see Figure 7.1).

A nice feature of the first prototype was that the model could be shown over the whole screen, replacing the Java Code as central element of the IDE. In the future prototype the elements could be arranged so that on the large full-screen diagram a corner can be left free, when the code is shown there.

Assign Methods from the Call Chain to Classes

In the current prototype the methods for the proactive navigation support are displayed isolated from their classes even if the classes to which the methods belong are displayed. The reason for that is that the classes might be located in an other position in the diagram due to other constraints (e.g. a super class should be located on top of the subclass). A more intuitive diagram could be achieved by two steps:

1. The methods should be grouped into types i.e. methods belonging to the same type should be displayed in the same box.
2. All types should be annotated with a unique property (e.g. a color code) so that the type-boxes in the call-chain can easily be associated with the types displayed in the diagram.

Screen Real Estate

Since developers use the IDE in a different way (with different windows open) and because they have different screen resolutions, the proposed grid layout could gain or loose cells depending on the size of the diagram window in Eclipse. For example if the the window has a large height but only a small width the could loose two columns (one on the left and one on the right). If the screen is set to full, there should be space in one corner for the code. To optimize this empty space, the size of the Java Editor should then equal the size of a whole number of sections.

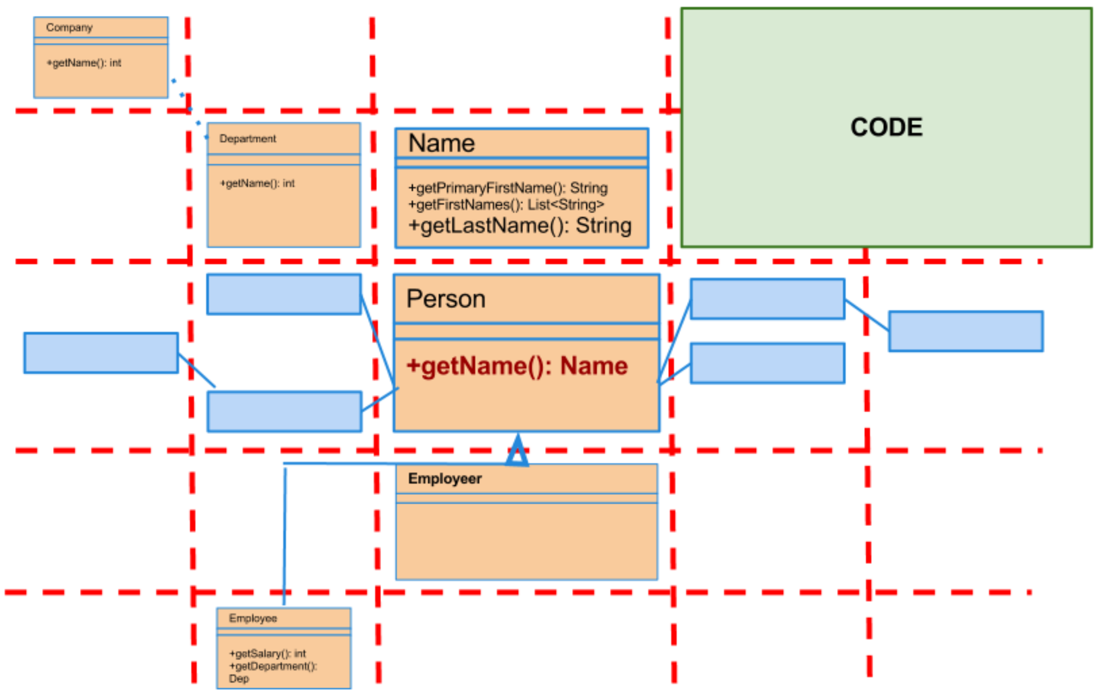


Figure 7.1: The layout of a possible future prototype when displayed in a maximized window: the call chain may contain more than direct callers and callees, there is a specific place left empty so that the source code editor does not overlap the diagram and the size of code elements depends on the relevancy value of the code element.

Chapter 8

Conclusion

We proposed an approach which helps developers navigating and understanding code by capturing the developer's interaction and visualizing the relevant code structure.

First, we have presented an exploratory study with twelve developers performing bug fixing tasks on three open source systems. The study shows that developers often revisit code when they explore the relevant code elements before they complete their task. It also shows that developers that use structural navigation aids from the IDE can complete the task faster. When the developers were asked to draw the essential code structure which they had to understand in order to complete the task, they only used few code elements (in average 4.1 classes).

Based on the study, we developed an approach that captures a developer's navigation steps, analyzes the underlying code structure, and determines relevant elements based on a set of heuristics. The relevant elements together with their structural dependencies are presented to the developer in a UML-like diagram. This diagram also shows a call chain which gives the developer proactive navigation support.

We implemented a prototype of the approach as an Eclipse plug-in. The diagram was implemented to be interactive and maps the code elements to the source code editor of the IDE. The prototype was developed in two iterations showing two different visualizations of the generated model.

We have evaluated the approach with the data of our initial exploratory study. We simulated the transcribed developer interactions which induced the prototype to generate the diagram after each step of interaction. The results show that in the simulations, the code elements that were revisited by the developers have been visible before the revisits happened. We think that the selection of code elements in the generated diagram is a promising compromise between the available screen real estate, which limits the number of displayable elements and the need of having all relevant code elements in the diagram.

Future work includes a more flexible diagram that shows more details when the diagram has more space in the IDE and resizes the displayed code elements according to their relevance.

Bibliography

[arc] Architexa, http://www.architexa.com/.

[BFVY96] Frank J. Budinsky, Marilyn A. Finnie, John M. Vlissides, and Patsy S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[BJR00] Grady Booch, Ivar Jacobson, and Jim Rumbaugh. Omg unified modeling language specification. *Object Management Group ed: Object Management Group*, page 1034, 2000.

[BRZ$^+$10] Andrew Bragdon, Steven P Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J LaViola Jr. Code bubbles: rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 455–464. ACM, 2010.

[bug] Bugzilla, http://www.bugzilla.org/.

[DAB08] Wojciech J Dzidek, Erik Arisholm, and Lionel C Briand. A realistic empirical evaluation of the costs and benefits of uml in software maintenance. *Software Engineering, IEEE Transactions on*, 34(3):407–432, 2008.

[DP06] Brian Dobing and Jeffrey Parsons. How uml is used. *Communications of the ACM*, 49(5):109–113, 2006.

[dra] Draw2d. http://www.eclipse.org/gef/draw2d/index.php.

[FP06] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in java. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 855–865. ACM, 2006.

[fre] FreeMind. http://sourceforge.net/projects/freemind/.

[FSB12] Thomas Fritz, David Shepherd, and Christoph Bräunlich. Supporting search and navigation through code context models. Technical report, 2012.

[gef] GEF: Graphical Editing Framework. http://www.eclipse.org/gef/.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. *ECOOP'93—Object-Oriented Programming*, pages 406–431, 1993.

[gua] Guava Libraries. https://code.google.com/p/guava-libraries/.

[jdt] Eclipse JDT. http://www.eclipse.org/jdt/.

[jpa] Java PasswordSafe. http://sourceforge.net/projects/jpwsafe/.

[jun] JUnit. http://junit.org/.

[KAM05] Andrew J Ko, Htet Htet Aung, and Brad A Myers. Eliciting design requirements for maintenance-oriented ides: a detailed study of corrective and perfective maintenance tasks. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 126–135. IEEE, 2005.

[KKD$^+$11] Thorsten Karrer, Jan-Peter Krämer, Jonathan Diehl, Björn Hartmann, and Jan Borchers. Stacksplorer: Call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 217–224. ACM, 2011.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97—Object-Oriented Programming*, pages 220–242, 1997.

[KM05] Mik Kersten and Gail C Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM, 2005.

[KM06] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.

[KMCA06] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *Software Engineering, IEEE Transactions on*, 32(12):971–987, 2006.

[LGHM07] Thomas D LaToza, David Garlan, James D Herbsleb, and Brad A Myers. Program comprehension as fact finding. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 361–370. ACM, 2007.

[LM10] Thomas D LaToza and Brad A Myers. Developers ask reachability questions. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 185–194. IEEE, 2010.

[LM11] TD LaToza and BA Myers. Visualizing call graphs. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pages 117–124. IEEE, 2011.

[MKRČ05] Gail Murphy, Mik Kersten, Martin Robillard, and Davor Čubranić. The emergent structure of development tasks. *ECOOP 2005-Object-Oriented Programming*, pages 734–734, 2005.

[myl] Mylyn. http://www.eclipse.org/mylyn/.

[rac] Rachota. http://sourceforge.net/projects/rachota/.

[RCM04] Martin P Robillard, Wesley Coelho, and Gail C Murphy. How effective developers investigate source code: An exploratory study. *Software Engineering, IEEE Transactions on*, 30(12):889–903, 2004.

[RM02] Martin P Robillard and Gail C Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th international conference on Software engineering*, pages 406–416. ACM, 2002.

[RM03] Martin P Robillard and Gail C Murphy. Feat: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823. IEEE Computer Society, 2003.

[SDRF12] David Shepherd, Kostadin Damevski, Bartosz Ropski, and Thomas Fritz. Sando: an extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 15. ACM, 2012.

[SKM06] Vineet Sinha, David Karger, and Rob Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 187–194. IEEE, 2006.

[SLVA10] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. An examination of software engineering work practices. In *CASCON First Decade High Impact Papers*, pages 174–188. ACM, 2010.

[stua] Artifacts of the Exploratory Study. https://github.com/abb-iss/study-artifacts-for-code-context-models/.

[stub] Study Artifacts. https://github.com/abb-iss/study-artifacts-for-code- context-models.

[zes] Zest: The Eclipse Visualization Toolkit. http://www.eclipse.org/gef/zest/.