Visualize local neighborhood for supporting debugging

Master's Thesis

Niels Cobben

Visualize local neighborhood for supporting debugging

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Niels Cobben
born in Delft, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Document layout formatted with WinEdt 5.4/MiKTeX 2.7.
Charts created with Microsoft Excel 2007.
Flow diagrams created with Microsoft Visio 2003.
Experiment documentation created with Microsoft Word 2007.

Visualize local neighborhood for supporting debugging

Author: Niels Cobben
Student id: 1099701
Email: N.R.Cobben@student.tudelft.nl

Abstract

Developing software is complex, debugging even more. In this thesis an approach is presented to reduce the debugger's burden by introducing visual support for debugging. This is accomplished by using multiple supporting debugging concepts which are implemented in the tool: the Visual Debugger. The Visual Debugger supports debugging by visualizing the local neighborhood at runtime when debugging with a specialized hierarchical graph. The visualization provides the user easy access to an overview and details of the dependencies surrounding the executing method. The concept of visualizing local neighborhood to support debugging is tested in a pre-post test user experiment in which participants solve debug tasks with the Visual Debugger and answer questions about their debug habits. The questions involve how participants experience the tool and how they use a debugger. The results show that the capability of the Visual Debugger to support debugging by visualizing the local neighborhood is useful but more optimization and research is required to relate this result to improved program understanding.

Thesis Committee:

Chair: Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor: Dr. M. Pinzger, Faculty EEMCS, TU Delft
Committee Member: Drs. P.R. van Nieuwenhuizen, Faculty EEMCS, TU Delft

Preface

Before you lies the document resulting from the research assignment in the field of Computer Science. Visible is the text and images explaining the research approach, invisible is the amount of effort and hours put in to create this. Without the help of other people this document could not be made. I want to thank Andy Zaidman for the advice in setting up the experiment; the participants of the experiment for dedicating their time, effort and for sharing their opinion about the experience with the Visual Debugger; and finally Martin Pinzger, for his support in the whole Master's project and reminding me to keep on track following the red line.

Niels Cobben
Delft, the Netherlands
October 20, 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software is an abstract concept. Software is created and developed until it reaches to a close approximation of its specification. This is a difficult process covering multiple domains. This thesis focuses on the debugging aspect of software development. This chapter states the problem and explains the approach.

1.1 Problem statement

Some programs provide the ability to run in debug mode. A debug mode usually means more output in a log file that is suitable for feedback for the developers. It is a final resort to solve bugs that have slipped through the testing phase. When a user of the program encounters a bug, the log file can be send back to the developers. The time interval between a bug is reported and the responsible code is written can be substantial. When debugging such bugs, the limited information and limited knowledge of the program can result in a long process of relearning the program's design and behavior. The problem can be described as: when dealing which unfamiliar software, how can debugging be supported?

1.2 Research approach

The problem is approached from three different perspectives. Debugging requires understanding of the program. The first perspective is program understanding. The second perspective is from the developer perspective. How do developers perform debugging? The third perspective is from the technical perspective: what techniques can be used to reduce the complexity in the hierarchical structure of software without losing its content? One aspect of the complexity of software is dependencies. In source code written with Java, different dependencies exists among its classes and methods (see Table 1.1). An additional complexity is that some dependencies are created (dynamic) at run-time and some are created (static) by compiling.

Table 1.1: Dependencies

Type	Dependency	Static/Dynamic
Method-to-method	calls	dynamic
Class-to-class	inheritance, subtyping	static
Method-to-class	instance (cast/instanceof/type, access)	dynamic

In Listing 1.1 a small example is made to show the dependencies (indicated by the green comments) in a class.

```
1  Class Car extends RoadVehicle implements Vehicle { //subtype RoadVehicle, inheritance Vehicle
2      Name carName = new Name("label"); //type Name, call new, type String
3
4      public void overTake(RoadVehicle otherCar){
5          if (otherCar instanceOf Vehicle) { //type RoadVehicle, instanceOf Vehicle
6              Vehicle targetCar = (Vehicle) otherCar; //type Vehicle, cast Vehicle
7              targetCar.inform(carName); //access targetCar, call inform, access carName
8          }
9      }
10     ...
11 }
```

Listing 1.1: Example class with different dependencies.

This small code segment already reveals the complexity related to the amount of dependencies.

1.2.1 Research questions

From these perspectives (program understanding, developer and technical) three research questions are constructed:

- **Question 1: Can local neighborhood visualization be used for supporting debugging?**

The research question is: if we provide context information for interesting locations (determined by breakpoints) in the source code, can this be used for supporting debugging? Providing instantaneous access to dependencies should reduce the need for searching those dependencies which supports debugging. This research question is answered with the answers of the two following research questions.

- **Question 2: What are key points for supporting debugging?**

Before actually attempting to solve a problem we have to look at the problem area. For supporting debugging what are the problems? The focus is on the debugger usage in which the following questions are asked:

- Which features are used of the debugger?

- Which criteria are used to determine when to inspect?
- How are objects inspected?

Debugging and program understanding are strongly related. This leads us to the next research question.

- **Question 3: Is the use of debugger for program understanding (general) *and* debugging (specific)?**

Can program understanding capabilities of a debugger be useful applied for purposes besides debugging? Supporting program understanding is useful in situations when dealing with unfamiliar software.

1.3 Thesis structure

What is the concept of visualize local neighborhood and how can it be used for supporting debugging? This concept lies in the domain of visual debugging. The background of this domain is explained in Chapter 2 by researching the underlying domains of program understanding, software visualization and debugging. In the first part of Chapter 3 the concept of visualizing neighborhood is explained as a combination of other concepts. To answer the question whether this can be used for supporting debugging a prototype tool named Visual Debugger is designed that has implemented the visualize neighborhood concept. The design of the Visual Debugger is described in the second part of Chapter 3. The implementation and architecture of the Visual Debugger is described in Chapter 4. An evaluation of the visualize neighborhood concept is accomplished by testing the Visual Debugger on debug tasks. The debug tasks are setup in an experiment to answer the research questions. The experiment's design is described in Chapter 5 and the results are described in Chapter 6. In Chapter 8 the conclusions are drawn. In this chapter also directions for future work are described. Techniques and methods related to the Visual Debugger are described in Chapter 7.

Chapter 2

Background

The scope of the research is defined in the first chapter; this chapter focuses on the background research. Preceding this thesis, the conducted literature study revealed insight in the underlying domains of visual debugging: program understanding and software visualization. Within these domains, different techniques are presented from (cognitive) psychology, graph theory and debugging. These techniques are used for the concepts to support debugging.

2.1 Program understanding

The first domain in the background research is program understanding. Debugging is performed to improve the current understanding of the inspected software. To improve debugging effective, it is required to understand the processes in program understanding.

To understand a program, a developer can perform three actions: read the available documentation, read the source code and inspect the program's execution behavior. The source code is usually the primary source of information. The source code is not a static entity. Due requests from different stakeholders the source code is continuously maintained until the support of the software is ended.

2.1.1 Software maintenance

Program understanding is usually associated with software maintenance. Lientz et al. show that a large part of working on software is in the category of maintenance [22]. Corbi describes challenges in program understanding including maintenance [6]. Maintenance can be divided into further subcategories:

- Corrective maintenance: correct a design flaw
- Adaptive maintenance: adjust to new hardware or other upgrades
- Perfective maintenance: enhancements for user requests

- Preventive maintenance: adjustments to improve future maintainability

This maintenance changes not only the system's functionality but also its design. Lehman found a relation between software evolution and software quality. When significant effort is required to understand the original design and the current 'design', the source code is deteriorated [20]. The deterioration is characterized by a design that ignores program structure, data and function abstraction. The original design did not change with the changes in the source code. Mens et al. called this kind of evolution co-evolution; it is a current challenge in software evolution [25].

2.1.2 Challenges

In program understanding there are more challenges besides software evolution. One aspect is the lack of or insufficient amount of documentation. Another problem is outdated documentation in which changes in the software are not (yet) documented. Rajlich describes several program understanding challenges [32]. One of the challenges is understanding the many-to-many relations inherent in software. Also the quantity of source code lines in software system presents a challenge. Miller shows that our short-term memory limits us only to accommodate 7 concepts at a time [26]. Because of this limitation, approaches in program understanding will require abstraction strategies.

2.1.3 Comprehension models

In studies of program understanding there are mainly three important cognitive models. The comprehension models can be divided into the applied strategy of program understanding: the top-down, the bottom-up (or chunking) and the mixed (or opportunistic) strategy [40]. The top-down model consists of mapping the user's knowledge, grouped in plans, to the source code to verify its own ideas and hypotheses. In this model the mental representation of the software is constructed from the highest abstraction level through the lower levels containing more detail. Research suggests is it mostly applied when dealing with familiar code. The bottom-up model is described as chunking up small parts to form bigger chunks. In this model the mental representation of the software is constructed from the small parts of high detail that are merged into bigger parts of lower detail. Research suggests that this strategy is applied when dealing with unknown code. The opportunistic model is described as a combination of the top-down model and the bottom-up model. Von Mayrhauser and Vans show that dependent on the comprehension task, the programmer switches between models [39].

Rajlich extended the opportunistic comprehension model with a scale property [32]. The scale property is defined as: on which scale is the program understanding performed. For small programs, program understanding can be accomplished for the whole program. For large programs, understanding the whole program is unrealistic. The opportunistic approach consists of searching for the desired concept in the software to understand. This results in a partial understanding of the program which makes program understanding less complex and

more feasible. Program understanding tools are effective when it should answer precisely the question rise during validating hypotheses [3].

2.2 Software Visualization

The previous section describes the program understanding domain and its challenges. For supporting debugging, supporting program understanding is required. In program understanding the complexity lies in the amount of elements and its slow comprehension process. This section follows an approach in the domain of software visualization. The area that is covered includes graph techniques and visual optimization techniques.

2.2.1 Advantages

Using visualization provides advantages over a textual representation of a source code. It provides solutions for reducing complexity by visual support. The complexity lies in the associations and amount. Current visualization techniques contain smart solutions for presenting hierarchical structured data. These visual presentations allow a high number of elements and associations. Further information compression can be achieved by combining elements in groups. Visualizations are better suited for extracting certain information: Dunsmore mentions the bandwidth of the human vision system, speed of tracking and detecting visual patterns and abstraction power in pictorial representations [8].

2.2.2 Graphs

Larkin and Simon show that graphs as visualization have a distinctive advantage over a textual representation of a source code [19]. In source code, dependencies must be searched, in graphs these can be made explicit. Another advantage is recognition: graphs provide some visual preprocessing by grouping information together. This results in the main advantage of graphs which is the ability to index information to support efficient processing. Different types of graphs have different strong visual points. Few gives examples of strong visual points for different graphs [10]: a strong visual point of a line graph is presenting the overall shape, a strong visual point of a bar chart is the ability to individually compare bars. The dimension of time is often used to represent a historical overview. Meaningful characteristics of change through time includes: magnitude, shape, velocity and direction. More attributes can be used to add additional information, for example using shapes instead of dots for a scatterplot.

Mutzel and Eades show that a graph which requires a small amount of effort to understand, should follow some guidelines [28].

- Unrelated nodes should not intersect
- Edge crossing must be avoided, planar graphs are preferred

- Avoid sharp edge bends, reduce the total length of bends. A solution is to use an orthogonal drawing style
- Understanding requires less effort when all edges are short
- Reducing the drawing area will result in easier identification of objects
- Represent hierarchy
- Represent symmetry and patterns

These guidelines are more complex to maintain regarding a large graph consisting of many objects. To reduce the amount of objects one approach is to use clustering. Clusters can be determined by the user or automatically based on local properties such as the number of neighbor nodes. Another approach for following the guidelines is to optimize the layout of the graph. This can be accomplished with a layout manager. A layout manager adds semantic information by positioning nodes in the graph. In research it is shown that user guided graph drawing is beneficial. An automatic layout algorithm that changes too many edges can destroy the current mental image of the representation. Another option for layout is to use force directed placement (FDP). With FDP, a graph is modeled as a physical system. Nodes have weights that represent the amount of repulsive force [2]. Between the nodes, arcs have an attractive force with a length equal to the amount of force. After a number of iterations an equilibrium sets the graph layout stable (to a minimum energy state).

2.2.3 Focus and context

Even for visualization the amount of visualized elements is limited. Not all details for large data sets can be shown. One visualization option is overview and zooming. With focus+context this consists of an overview and a detailed view of data in one view. Häuser shows several focus+context techniques [14]. Focus is defined as a visualization technique to differentiate a subset of data. Focus is used for multiple purposes, we describe two. Purpose 1 is to provide more detail. One approach is to distribute more screen space to a subset of the data at cost of the remaining data. An example of this visualization technique is 'magnifying glass' in which a space-distortion technique is used to provide a higher detail for a subset of the data. Purpose 2 is to provide classification of data elements for easy distinguishing parts. This can be accomplished by using different colors or multiple levels of opacity. An example is highlighting elements by using a lighter color or a lower opacity level. Haber shows another approach to distinguish elements: the impression of depth [12]. This simulates the difference in distance of objects from the viewer. It is accomplished by using visual perception cues such as occlusion (partial hidden objects), transform (shear, shrink), shading and texturing.

2.3 Debugging

In a survey questioning solutions for understanding a large program, multiple categories of tools were provided as answer [33]. The category of 'code navigation' was listed first

and the category of 'debugger' third. Both categories of tools provide usable aspects for visual debugging. This section presents challenges for debugging. The goal is to enhance debugging with visual detailed information.

2.3.1 Debugging scope

Hailpern and Santhanam show that debugging is typical required during three activities in the development process [13]. First, when writing source code from design. Second, in the testing phase when unintended behavior is detected. The third phase is when the software system is in production or deployment. Debugging in later stages requires deeper analysis before problems can be found.

2.3.2 Debugging challenges

Eidenstadt focused a user survey on debugging: why is it hard to find bugs, how bugs were found and the root causes of bugs [9]. The first question is about the difficulties to find bugs. Half of the difficulties originated in two causes: large causality distance and bugs interfering with debug tools. The top three difficulties include:

- Bugs occur infrequent or inconsistent
- Tools cannot be used (too long to run, limited by memory)
- Fault assumption

The second question is about how bugs were found: the most dominant bug finding technique was reports of data-gathering (print-statements) and hand-simulation. Other techniques included stepping with debugger, dump and compare core dumps, use conditional breakpoints, use memory leak and references check tools, take a break or study code, spotting by expert and controlled experiments. The third question is about the root causes of bugs. The two biggest causes were memory overwrites and vendor supplied hardware or software faults. Other root causes included faulty design logic, wrong variables initialization, used wrong variable, syntax errors (typo's), unsolved bugs and language interpretation mismatch (assuming k means 1000 instead of 1024).

2.3.3 A visual approach

The combination of program understanding and software visualization is logical for visual debugging tools. Finding a good solution for presenting a view of the software that is in detail where required without getting lost, is difficult. The tool DA4Java uses a combination of techniques to optimize its visualization [31]. The visualization consists of a graph to represent the source code. This graph is a nested graph which benefits from the software hierarchical structure: methods and variables are grouped in type nodes and types are grouped in packages nodes. The nodes are connected through dependencies such as calls, inheritance and casts. The tool also provides graph editing capabilities such as the adding and removing of nodes and edges. The editing selection of the edges can be based on the

type of dependency, for example: a selection of all incoming calls of a node. The underlying framework of the tool provides the source code conversion and persistence options.

2.4 Summary

The background chapter describes different domains concerning visual debugging. From the program understanding domain, the complexity of understanding software is described. Developers approach this complexity by constructing a mental model with a particular strategy. The software visualization domain contains visualizations capable of showing complex structures. Graphs are a good solution to represent dependencies that characterizes software. For optimizing the amount of information, focus+context techniques can be used to balance multiple levels of detail. A combination of techniques and methods from program understanding and software visualization is required for successful visual debugging. Bugs can be challenging to find and can remain undetected until the software is in production. Designing a tool to visual support debugging contains a lot of challenges.

Chapter 3

Concepts and approach

The first half of this chapter introduces the supporting debugging concepts which provide the base for the supporting debugging approach. The second half of this chapter explains the approach and setups a design for tool support.

Supporting the development process with a new approach in debugging is no easy task. Current debuggers already provide debugging support with much functionality. The method used for debugging is dependent on the problem. For simple cases, a print statement of some object status is sufficient. For debug cases where the cause is not clear there are more rigorous options available such as logging and tracing. These methods require much user interaction before they provide useful results. Instead, a debugger provides a wide range of options for supporting different kind of debug problems. The new approach consists of extending the debugger with visual information and will be explained by defining multiple concepts for supporting debugging.

3.1 Concepts

This section describes the concepts for supporting debugging. The concepts are based on static analysis of the source code for providing dependency information instantaneously when debugging. Figure 3.1 shows the concepts in an overview. The main concept is visualizing source code. For additional debug input, communication with the debugger is added. Synchronizing through the automatic navigation a 'neighborhood' can be defined. Combined with the focus concept, this information results in adjusting the graph visualization for an optimized view to support debugging.

3.1.1 Visualize source code for supporting debugging

The first concept consists of visualizing source code. Debugging consists of discovering values within objects and dependencies between objects. An object `A` is dependent on another object `B` when a modification of object `B` can affect object `A`. A typical debugger only provides status information of objects at runtime. The dependencies need to be discovered by other means. In the Eclipse platform this information can be retrieved with "open type

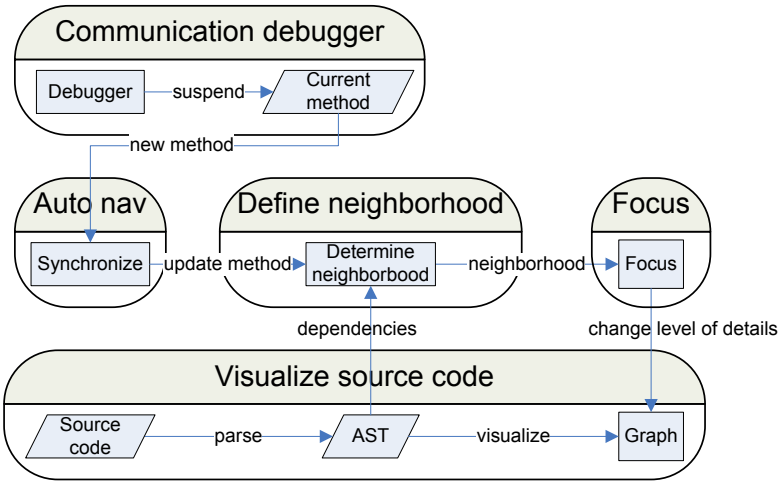


Figure 3.1: Overview of the supporting debugging concepts

hierarchy" and "open call hierarchy" commands. The limitation of these commands is that they work on a selection of only one type or call at the time. Another limitation is that the commands only work in one direction: there exists no command for showing which methods will be called by one method. The following concept provides a solution for the problem. Instead of letting the user requests the dependency information based on types and method calls, present this information automatically in a visualization. The idea is to show a graph presenting the source code to show the dependencies between methods. This provides an overview of the dependencies between methods which provides an easy way to analyze dependencies of objects. The graph can represent methods and types by nodes and can represent dependencies by arrows. For support of type and call hierarchy, dependencies can entail calls and inheritance.

3.1.2 Communication with the debugger

In the previous source code visualization, searching for the right methods is still required. The following concept provides a solution for that problem. When debugging, the program's execution is followed. With information from the debugger it is possible to visualize the current executing method. This allows for an easier way of tracking the program's execution and its surrounding dependencies. To support this kind of debugging it is required to access dynamic runtime information from the debugger. An implementation is explained in Section 4.2.2. When working in a debug session the debugger keeps track of the program it debugs. The debugger does not track the program itself but is notified when the debugger should be suspended. The suspending occurs when a breakpoint is reached or the program is interrupted by user interference. In a state of suspension, the debugger can be accessed for dynamic information of the thread that is suspended. This is presented in the form of a stack frame which contains the current method calls on the stack. This information can be retrieved to visualize the current executing point in a graphical view of the source code.

3.1.3 Automatic navigation

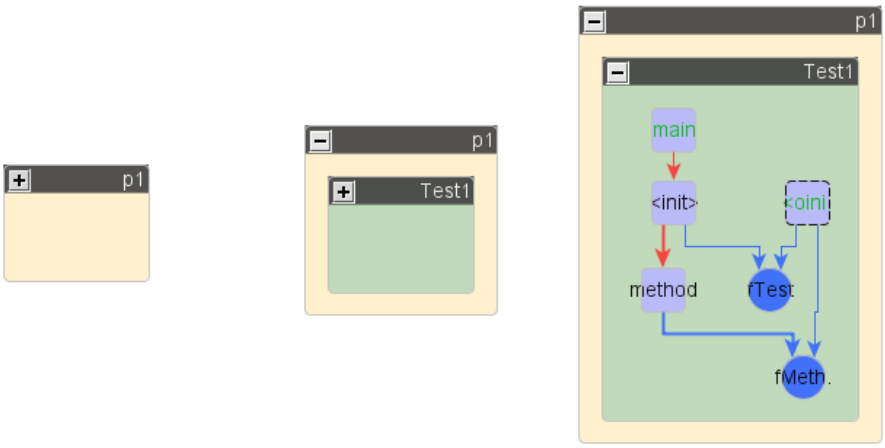
Many elements in the source code graph which are interactive (see Section 2.3.3) provide options to change the view of the graph. One of those options is the ability to change the level of detail. When an element has this ability, it can hide (or show) his inner elements to provide more (or less) detail. The less detailed version of an element provides more screen space for other (high detailed) elements. The automatic navigation concept consists of automating the selection of elements to hide and show. It provides the possibility to simulate navigation in the graph. When debugging a program, this concept will allow to update the graph with showing the high detail element of the current executing method and showing other elements in low detail. This should result in a decrease of the user's burden of navigating between the graph elements. This concept is accomplished by retrieving the current executing method (see Section 3.1.2) and controlling the detail level of graph elements.

3.1.4 Defining the neighborhood

Only visualizing the current executing point in a graph does not provide much more information than the debugger already provides, but showing the dependencies with other methods and types does. The neighborhood is defined as the incoming and outgoing dependencies of a particular method, other elements in the graph are defined as 'outside' the neighborhood. The classes and optional packages of that method and its dependencies are also included in the neighborhood. This definition will allow to define parts of the graph of particular interest in the source code. For an example see Section 3.1.6.

3.1.5 Focus mechanism

The graph representation of the source code can be structured by using the source code hierarchy. This hierarchy can be used to incorporate multiple levels of detail. The highest level of detail contains the content of types: methods and variables (image (c) in Figure 3.2). The next lower level of detail, contains the types (image (b) in Figure 3.2). The lowest level of detail, contains the packages (image (a) in Figure 3.2). With the multiple levels of detail it is possible to create a focus mechanism to show some graph parts, for example the neighborhood, in higher detail than other parts. In this way the available screen space of the graph can be optimized. Several focus mechanisms were considered. The first variant was without focus: all source code was represented in high detail. Except for toy programs, this would cost too much screen space to display all the graphical elements. The second variant was that only the neighborhood was showed and other elements of the source code were filtered. This provided detailed information but lacked the overview in relation to the remaining source code. A choice is made in a third variant which is a compromise of the first two variants: it shows the neighborhood in high detail and the remaining source code in low detail. In combination with the auto navigation, the focus on the neighborhood is updated when the debugger is suspended.



(a) Detail level 1 (b) Detail level 2 (c) Detail level 3

Figure 3.2: Multiple levels of detail in a nested graph.

3.1.6 Customization

How a tool is optimal used, is dependent on its user. In addition, users have different preferences of settings. For an increased usability of a tool, certain customization is required. Two customization options are described.

Dependency setting The first customization is on the neighborhood. The value of its setting indicates the size of the neighborhood. A size of value 0 indicates a neighborhood that consists of only the target method and no dependencies. Figure 3.3 shows that a size of 1 indicates the target method and all direct incoming and outgoing dependencies: method B `main` [incoming dependency]→ target method A `main` [outgoing dependency]→ method C `<init>`. Figure 3.4 shows a size of 2 which is defined as the size of 1 plus another level of dependencies: methods ABC and the incoming dependencies for method B and outgoing dependencies for method C. In this way the size of the neighborhood is defined and customizable as levels of dependencies. Showing the dependencies between all the methods in a graph requires much screen space while only a part is interesting when debugging.

Filter setting The second customization is the filter. This is an extension to the focus concept. The focus concept would still show elements outside the neighborhood in some detail. In situations where the neighborhood already takes up most of the screen space or contains all the parts of current interest, filtering 'outside' elements should be automated. This filter setting is able to filter the elements 'outside' the neighborhood in different levels: package, class and method. The package level filters all packages that are not defined by the neighborhood (see the black lined box in Figure 3.5). This is the lowest filter setting. The following filter settings are each progressive on the previous filter level. The next filter level is the class level. This setting filters the classes not defined in the neighborhood (blue lined boxes in Figure 3.5). Some 'boundary' packages contain classes of both inside and outside

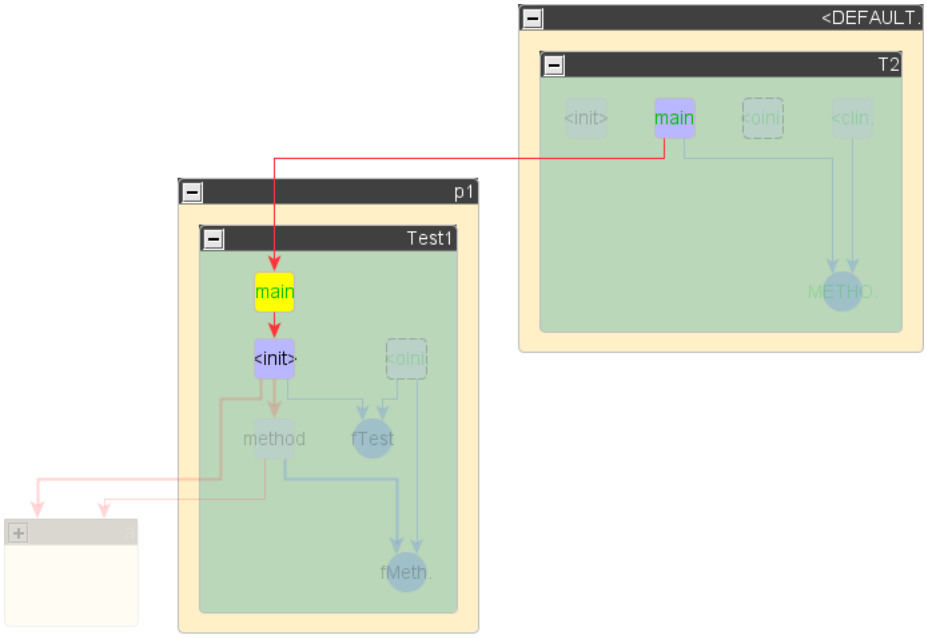


Figure 3.3: Neighborhood with dependency level of 1

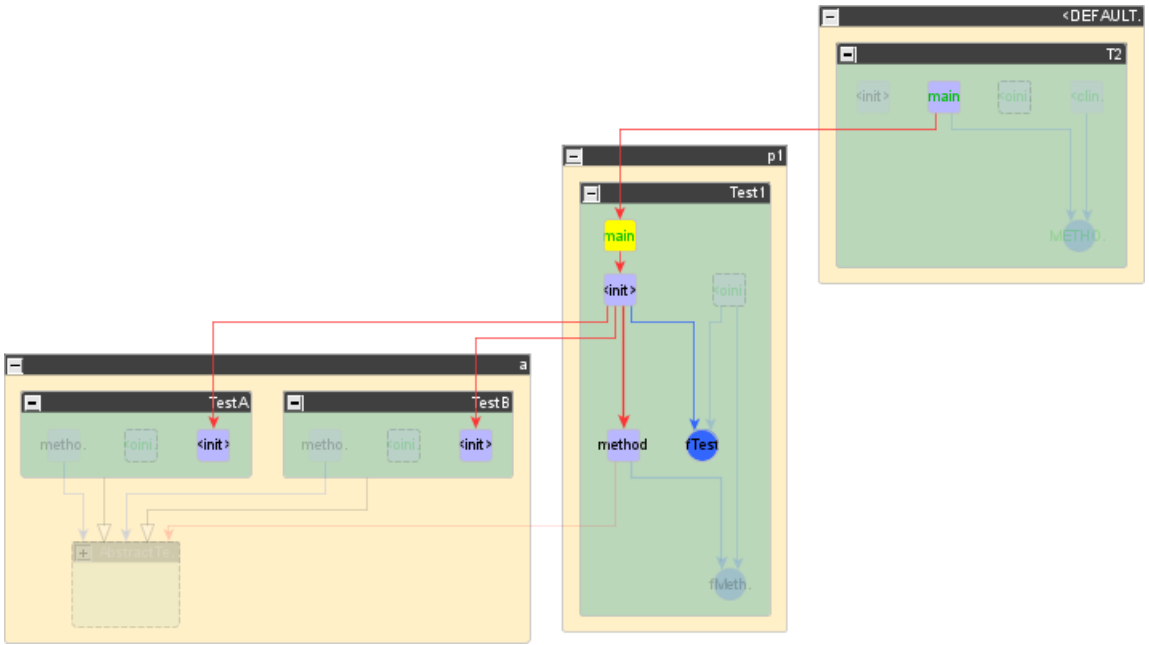


Figure 3.4: Neighborhood with dependency level of 2

the neighborhood. The filter setting ‘package’ will not filter these packages, but the filter setting ‘class’ will filter the ‘outside’ classes in these packages. The highest filter setting is

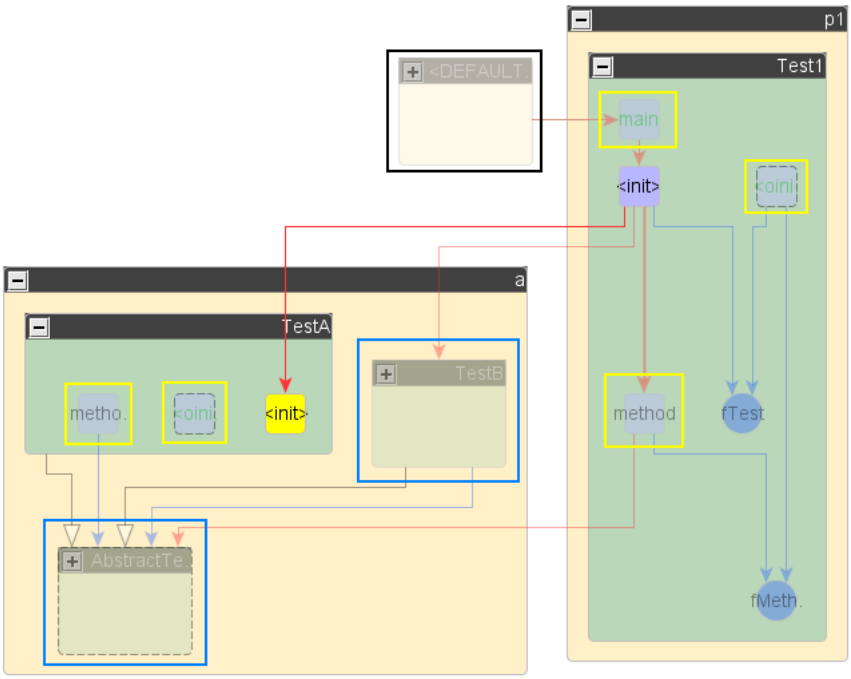


Figure 3.5: Neighborhood is defined by the two 'init' methods and their classes and packages. Different filter levels are indicated by the colored boxes

the method filter setting. This setting filters the 'outside' methods of 'inside' classes (yellow lined boxes in Figure 3.5).

3.2 Approach

With the concepts explained, the next step is to construct a design plan, which begins with a list of requirements. From there, an implementation is constructed to create a prototype tool implementing these concepts. The concepts are tested though an experiment designed for debug tasks. The results from that experiment are evaluated to conclude if the concepts are useful for supporting debugging. The analyses from the results provide the input to answer the research questions.

3.2.1 Requirements

Requirements are created before the design. The requirements are separated into functional and non-functional requirements.

Functional requirements The main functionality consists of the following items:

- Build a dependency graph from the source code

- Graph must provide the ability to show multiple levels of details: package, class, method
- Run-time modification support of the graph (able to add/remove nodes and edges)
- Retrieve information from the debugger (stack trace, suspend state)
- Access to settings for customization

To provide a visualization of the source code, it has to be transformed from a textual representation into a graph representation. For an optimization of the screen space, the graph elements require the ability to show different levels of detail. In addition, the graph should provide some customization options to show a partial graph. Besides customizing the graph, also a partial graph can be automatically constructed by using the runtime information received from the debugger to reflect the current executing method of the debugged program.

Non-functional requirements The requirements of the tool lies not only in its functionality. The non-functional requirements are the other big part of developing software.

- Fast - Actions issued by the user should not be tedious long for the program to execute
- Response - Feedback should be clear to acknowledge user's actions
- Usability - Deploy tool as plug-in

The tool GUI handling should be intuitive and commands should not taken a long time to execute. This will make the tool more likely for users to adapt to. To accommodate developers the whole tool should be integrated as plug-in for an IDE.

Constraints In the development it is also important to setup the constraints beforehand.

- Program language - Java
- IDE - Eclipse
- Focus on functionality

The programming language in which the tool is programmed is Java. The choice is based on the familiarity of the program language. The tool should be made as an IDE plug-in. This provides access to existing technology. The choice for the IDE to integrate the tool is Eclipse.[1] The main focus of the prototype tool is to include all concepts as functionality. Decorations such as nicer color schemes for the GUI should be a lower priority.

[1] `http://eclipse.org`

3.2.2 Tool design

From the requirements it is clear that the tool should contain a graph building component and a source-code-to-dependency-structure-conversion component. Instead of writing those components from scratch, a search to an already existing solution was conducted. The search eventually led to the tool DA4Java (see Section 2.3.3). This allows to delegate some part of the graph visualization.

The next step is the construction of a components design for the Visual Debugger. Figure 3.6 shows that the tool has the role as middleware between DA4Java and the debugger. The debugger's role is to provide runtime information consisting of the current executing method. The connection with DA4Java is two-ways: one for retrieving static information of the source code dependencies and one for controlling the graph visualization. The control component of the Visual Debugger contains the main functionality which includes auto navigation and determining the neighborhood. The output of the control component is subjected to the filtering system which filters and focuses graph elements. Customization of the filtering and neighborhood is accomplished through the settings component. It provides access for the user to change settings.

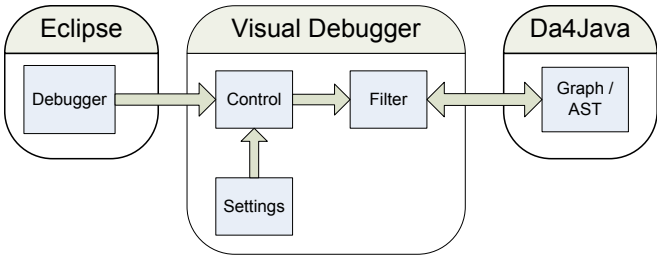


Figure 3.6: The components and connections design of Visual Debugger

3.3 Summary

This chapter explains the supporting debugging concepts of the Visual Debugger. The combination of the concepts leads to a system thats supporting debugging and program understanding. The first concept is visualizing source code. This is supporting debugging by providing an instantaneous view of the dependencies between methods and types. The concept of communication with the debugger involves the debugger suspend mechanism. It provides run-time information about which method is the current executing method. In combination with user set breakpoints it is used for the input of the Visual Debugger. The concept of auto navigation describes how showing or hiding different details of graph elements provide automatic navigation. The concept of neighborhood describes the dependency context of a method and its type. The focus mechanism provides a screen space optimization in which neighborhood elements are shown in high detail and other elements in low detail. The customization concept introduces the neighborhood and filter settings. From the concepts and

the requirements the component design of the tool is constructed. The design shows a separate component for visualization, debugger and control. Each component's role and each connection is explained.

Chapter 4

Implementation

From the supporting debugging concepts and design, the implementation of the Visual Debugger is started. This prototype tool relies on other tools and their techniques. First, an overview is presented covering those other tools and techniques. Then, the integration with Eclipse is explained in detail. The implementation details of the supporting debugging concepts are explained in a work flow. The implemented functionality of the tool is validated by performing unit testing.

4.1 Structural Overview

The Visual Debugger is modular build using other components. This section presents an overview to explain the role of those components.[1] The Figure 4.1 shows the composition of the components: Evolizer, Hibernate, DA4Java and yFiles. Evolizer's most important purpose is to provide the Famix model, for an easy access data structure representing the source code. Hibernate provides persistence options for performance. yFiles is a framework for creating graphs and DA4Java connects the Famix model with the graphs to manipulate the view.

[1] By using existing techniques software development can be more efficient: there is no use inventing the wheel twice.

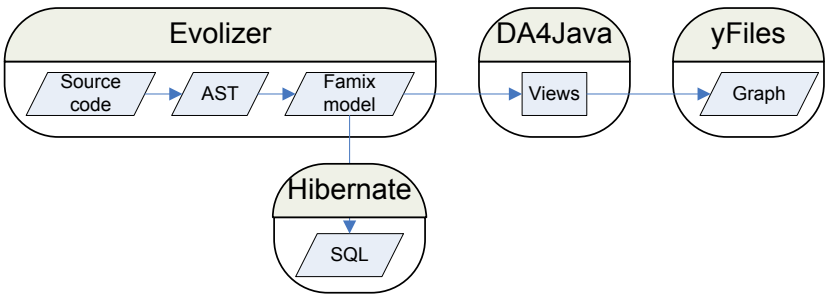


Figure 4.1: Dataflow within the used techniques

4.1.1 Famix model

Famix is a part of the open source platform for software and data analysis: Moose.[2] Famix is a family of meta-models representing models related to facets of software systems. The Famix importer of Evolizer[3] is used is to model static information of OO systems. The basis components of the Famix model consist of entities and associations. The entities represent Java language constructs such as: packages, classes, methods and attributes. Each Java construct has its own Famix entity implementation. The associations consist of invocations, accesses, inheritance, subtyping and cast/instanceof. Both entities and associations are annotated with properties such as static and abstract.

4.1.2 Hibernate

Source code can contain many entities and associations. For small and middle sized programs the computer's memory is sufficient to contain the whole source code representation. For larger programs other means of storage are required. A well-known solution is using disk storage through a database. It provides optimized access to structured data. For Java there are many possibilities. In the Evolizer tool such a database solution is already present: the open source software Hibernate.[4] Hibernate is a relational / object mapping software to create a connection with a SQL or H2 database. A H2 database is the in-memory variant of the file based SQL database. Hibernate provides options to create a mapping based on Java source code with annotations. These annotations consist of persistent and relations settings between objects such as one-to-many. Also cascade options are available. Cascade options allow an operation to propagate the hierarchical data structure in which the parent forwards the operation towards its children, for example remove a `Book` object will remove its (child) `Chapter` objects.

4.1.3 yFiles

The visualization of the Visual Debugger consists of a nested graph, a visualization solution from yFiles.[5] yFiles is a commercial library for structural graphs and diagrams. It provides the ability to automatically structure the layout of a graph. To reduce the amount of impact of restructuring, yFiles provides incremental layout which allows for a partial restructure of the graph layout. In addition, graph editing operations are supported. Care is taken to provide a consistent graph: deleting a node will cause the deletion of related edges (but deleting edges without deleting associated nodes is also possible).

4.2 Eclipse integration

The previous section describes the components that the Visual Debugger uses. This section describes how The Visual Debugger is connected with the IDE Eclipse. The integration

[2] `http://www.moosetechnology.org/docs/famix`
[3] `https://www.evolizer.org/wiki/bin/view/Evolizer/Features/Famix`
[4] `http://www.hibernate.org`
[5] `http://www.yworks.com/en/products_yfiles_about.html`

consists of the starting point of the tool, the input from the debugger and the configuration page.

4.2.1 Plug-in connection

Eclipse is well known for its extensibility which shows in the amount of plug-ins available on the internet. For the integration of the Visual Debugger with Eclipse, one of the many online plug-in tutorials [38] is used. For the plug-in connection with Eclipse the plug-in API[6] is used. In Eclipse, particular parts can be extended through the use of extension points such as editors, menus and actions. Plug-in connections are configured in a plug-in XML configuration file. Eclipse provides a GUI for easy access to edit the configuration file. The plug-in can be tested through the launch of the 'Eclipse Application' command which launches another Eclipse instance with the plug-in loaded.

The first point of integration of the Visual Debugger with Eclipse is the launching point. Because the tool is designed to debug one project at the time, the launch point is accessible from the package explorer context menu. The package explorer allows the user to select different resources (projects, packages and classes) to start the Visual Debugger with. In the plug-in configuration the following extensions are added:
`org.eclipse.ui.commands debug.showGraph` (provide command id)
`org.eclipse.ui.menus popup:org.eclipse.ui.popup.any?after=additions` (add menu item)
`org.eclipse.ui.handlers debug.showGraph,debug.command.ShowGraphCommand` (map command id to launch point in the plug-in)
When the launch command is executed the visualization of the Visual Debugger is started and the Eclipse perspective is set to 'Debug'.

4.2.2 Debugger connection

The second point of integration of the Visual Debugger is the connection with the Eclipse debugger. The extension capability of the Eclipse debugger is broad: it can be used for other programming languages, for example a C debugger for embedded systems [29] or Prolog [18]. Writing an (extension of the) debugger[7] is one approach. But for the purpose of the Visual Debugger, extracting run-time information is sufficient. This was accomplished by using the Platform plug-in API.[8] First, a listener is added to the Eclipse debugger:

```
DebugPlugin.getDefault().addDebugEventListener(MyListener);
```

Second, only check the suspend debug events:

```
DebugEvent.SUSPEND
```

[6] `http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html`

[7] `http://www.eclipse.org/eclipse/debug/documents.php`

[8] `http://help.eclipse.org/galileo/topic/org.eclipse.platform.doc.isv/reference/api/overview-summary.html`

A suspend event is triggered by the following options: when a breakpoint is hit, a stepping action has returned or by client request (hitting the debugger's 'pause' button).

4.2.3 Property page

In addition to the launch point, also the property page is constructed with the plug-in API. This GUI page provides access to set the preferences of the Visual Debugger. In the plug-in configuration, the property page is added by using the extension: `org.eclipse.ui.propertyPages`. The GUI components (selection box, buttons) are build from components of the SWT framework.[9] The Figure 4.2 shows the settings based on the supporting debugging concepts: neighborhood, filter and focus (transparency). The settings are stored on project basis.

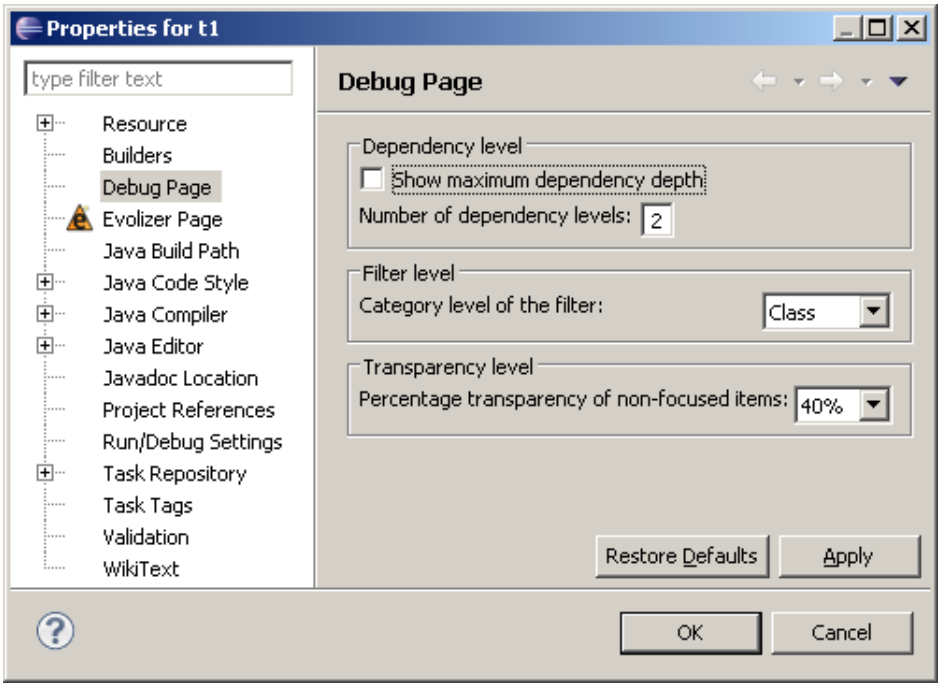


Figure 4.2: The property page of Visual Debugger

4.3 Work flow

With the external links (Eclipse and other tools) of the Visual Debugger covered, this section describes the internal work flow. The work flow is divided into three phases:

[9] `http://www.eclipse.org/swt/`

1. Executing method	2. Neighborhood	3. Graph
Receive debug event	Define neighborhood	Apply focus on neighborhood elements
Extract executing method		Apply filter on neighborhood elements

4.3.1 Extracting the executing method

Section 4.2.2 shows how the debug event is received from the debugger. This event contains information about the suspension status of the debugger. The sequence of extracting the current executing method from a debug event is as follows:
`debugEvent.getSource();` returns `IJavaThread`
`thread.getTopStackFrame();` returns `IJavaStackFrame`
The method identifier is constructed from the stack frame:
`stackFrame.getMethodName();`
`stackFrame.getDeclaringTypeName();`
`stackFrame.getArgumentTypeNames();`
With the current executing method extracted, we now can determine its neighborhood.

4.3.2 Determining the neighborhood

The Famix entity representing the executing method is retrieved from the Famix model through a Hibernate session: `fQuery.queryEntitiesByUniqueName(name);`. The next step is determining the neighborhood of that method by retrieving the incoming and outgoing dependencies.

```
1  private void getDependencies(AbstractFamixEntity method) {
2          ...
3          for (; ((level < fGC.getDependencyLevel()) || fGC.getMaxDependency()); level++) {
4              ...
5              // in
6              fQuery.queryDependentEntities(entities, dependentEntities, AbstractFamixEntity.class,
                       FamixAssociation.class, fTo, 0, 1);
7              fCMethodIn.add(dependentEntities);
8          }
9          ...
10         for (; ((level < fGC.getDependencyLevel()) || fGC.getMaxDependency()); level++) {
11             ...
12             // out
13             fQuery.queryDependentEntities(entities, dependentEntities, AbstractFamixEntity.class,
                       FamixAssociation.class, fFrom, 0, 1);
14             fCMethodOut.add(dependentEntities);
15         }
16     }
```

Listing 4.1: Retrieving dependency methods.

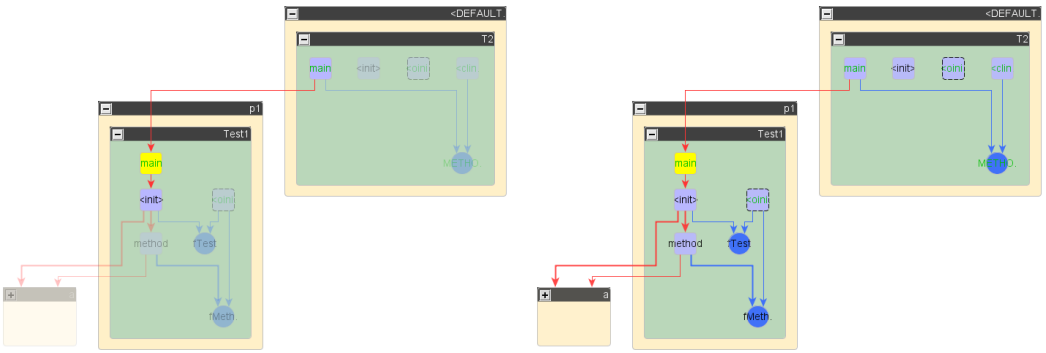
The Listing 4.1 shows that the dependencies are retrieved by level and divided into an in-set and an out-set. This dividing is required for the focus concept explained in the next section.

Because this method only retrieves the neighborhood methods, the remaining neighborhood classes and package must be retrieved. This requires no Hibernate session because the parent-child association is already present in the Famix entities. The next step is passing the selection to the graph.

4.3.3 Update the graph

The neighborhood selection is compared to the selection of the previous update. This way it can be determined which nodes should provide more detail and which nodes should provide less detail. The change in details is accomplished by 'opening' and 'closing' nodes. The result is that the previous neighborhood is changed to low detail and the current neighborhood changed to high detail. This editing of the graph is performed careful to not disturb the mental representation of the source code. To accommodate this, only nodes that were previously automatically 'opened' (because they were a part of the neighborhood) are 'closed'. Nodes that are manually opened by the user to investigate the graph, remains opened after a neighborhood update.

Before the nodes are 'opened' or 'closed' the filter is applied. According to the concept (see Section 3.1.6), the determined filter selection is removed from the graph. Then, the Visual Debugger checks if the selected nodes are present in the graph. When they are filtered, they are re-added. Dependent on the neighborhood setting some elements are outside the neighborhood. The visualization technique of highlighting distinguishes this separation. This is accomplished by applying transparency for the outside neighborhood elements (see Figure 4.3). To set the transparency, the nodes and edges are separately handled. This is



(a) Focus enabled (b) Focus disabled

Figure 4.3: Focus of the neighborhood by using transparency.

because the neighborhood is defined by the Famix entities and associations. By using layers of in- and out sets (see previous section), the associations are retrieved by determining the associations in one direction between sets (see Listing 4.2). Besides the neighborhood's nodes and edges, its executing method is distinguished by using a bright color yellow.

```
1 private void addNewEdgeFocus(int level, List<Set<AbstractFamixEntity>> list, SnapshotAnalyzer
      query, String direction) {
2     for (int i = 0; i < level; i++) {
3         set1 = new ArrayList<AbstractFamixEntity>(list.get(i));
4         set2 = new ArrayList<AbstractFamixEntity>(list.get(i + 1));
5         result = query.queryAssociationsBetweenEntitySetsOneWay(set1, set2, FamixAssociation.
              class, direction);
6         ...
7     }
8 }
```

Listing 4.2: Retrieving associations from entity sets.

4.3.4 Optimizations

Manipulation of a data structure that is a representation of a source code easy leads to performance issues. The following solutions are implemented to optimize the amount of data processing:

Batch-processing When an operation is performed on a set, combine items to fill the set. The reduction in the total amount of sets, reduces the overhead of the operation. In the implementation this is used for combining the nodes of the neighborhood to 'open'. The re-layout operation of the graph is now applied after a neighborhood change instead of a node change.

Caching When accessing the same data twice, caching the data will prevent performance loss by skipping the calculating part of that data. For example: the cache of the neighborhood representation in Famix entities. Because this data is accessing twice: one for visualizing it in the graph (when it is new) and one for removing it from the graph (when it must be updated).

Optimizing calculations For example: when the filter is applied, the non-neighborhood associations has to be determined which can be computative intense. Assume that the neighborhood is a small part of the full graph. To determine the non-neighborhood associations, get all (graph) associations. Determine the remove set by removing the neighborhood associations from the graph associations. The neighborhood associations can be determined from the neighborhood entities. This is faster than determine the non-neighborhood associations from the non-neighborhood entities.

4.4 Testing

To ensure correct behavior of the Visual Debugger, unit testing was applied. It was used to verify functionality and consistency. But this approach caused some problems: most functionality of the tool could not be easily separated from each other. Instead of testing functionally separately, multiple functions were tested simultaneously (see Figure 4.4).

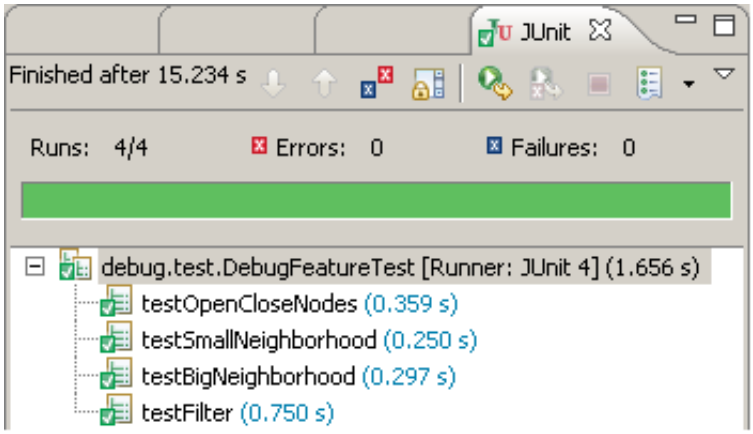


Figure 4.4: Results of the JUnit tests

Preferable, the tests should validate all possible scenarios, but even with discarding different settings this was already unfeasible. A solution was found in a selection of interesting boundary values for the parameters and a simulation of a debug session to validate the expected behavior. The simulation consisted of 'changing the executing method' to change the neighborhood. The tests revealed a few minor bugs which were fixed.

Dependency test: Validate different dependency settings. The settings were tested for only incoming nodes, only outgoing nodes and both.

Transparency test: Validate if the neighborhood entities were highlighted. The highlights are implemented as setting the neighborhood entities non-transparent and the other nodes transparent.

Filter test: Validate if the graph has filtered the correct entities.

Node open/close test: Validate different open/close scenarios. When updating the graph, nodes are required to open and close in order: first handle the top node and then lower (inner) nodes for an open command, for a close command it is the other way around (see overview in Table 4.1).

Table 4.1: Test scenarios for opening and closing graph nodes

Close nodes		Open nodes	
close package	close class	open package	
	close class	open package	open class
close package			open class

4.5 Summary

This chapter provides the implementation details to implement the supporting debugging concepts in a prototype tool. The Visual Debugger uses other tools and techniques: the Famix model from Evolizer, Hibernate to query dependencies from the source code and yFiles for a graph visualization. Besides these tools, the Visual Debugger also uses Eclipse technology: it launches from the package explorer, subscribes to debug events and sets preferences in a property page. The internal work flow of the Visual Debugger explains how the neighborhood is extracted from a Hibernate session and how the visualization is optimized to visualize this neighborhood. The Visual Debugger's performance issues and unit tests are also explained.

Chapter 5

Experiment design

With the implementation of the Visual Debugger completed, the concepts can be evaluated to answer the research questions. The first research question is: *Can local neighborhood visualization be used for supporting debugging?* For this research question the evaluation of the supporting debugging concepts is accomplished by validating the Visual Debugger with a pre-posttest experiment. Validating the Visual Debugger requires testing its functionality for a realistic debug problem. The second research question is: *What are the key points for supporting debugging?* For more insight in supporting debugging, research in the debugger usage is required. The evaluation will be focused on the debug features, breakpoints and object inspection. The third research question is: *Is the use of debugger for program understanding (general) and debugging (specific)?* A comparison is made between usability of the Visual Debugger for program understanding and for debugging.

The evaluation is accomplished by setting up an experiment. The goal of the experiment is that its results provide answers for the research questions. This chapter explains the design of the experiment including: preparation and setup.

5.1 Preparation of the experiment

In the experiment the Visual Debugger is evaluated on a test project. The role of the Visual Debugger is to support debugging the test project. The choice for which test project is explained in the following section.

5.1.1 Selecting the project for the experiment

The selected test project for the experiment has the following properties: practicality, complexity and representative. With practicality is meant that the project is used. The users can provide feedback from using the project which can improve further development. The second property is complexity. A project that contains complex source code such as a deep inheritance structure, allows for evaluation of the capability of the Visual Debugger to support program understanding. In the extension of the previous property complexity, the third

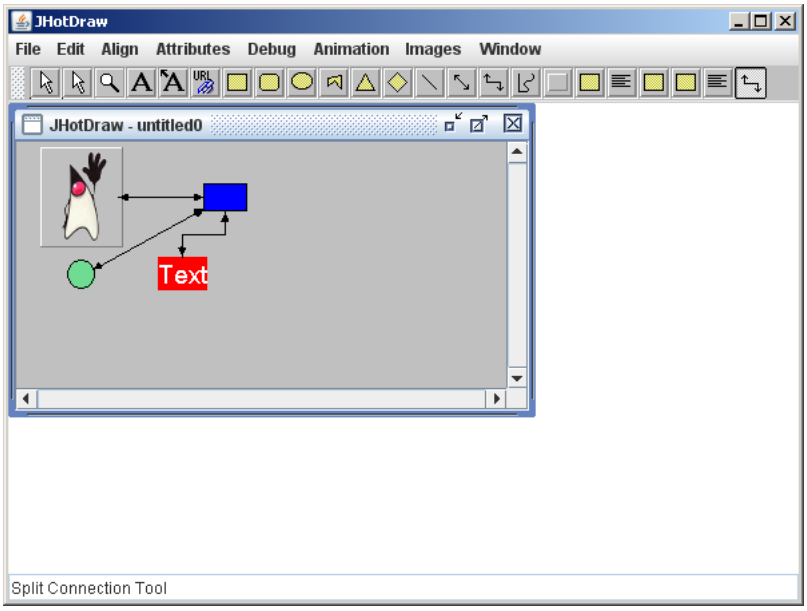


Figure 5.1: JHotDraw drawing program

property is representative: a project should be representative in comparison with commercial projects. This is accomplished by choosing a project of considerable size (between 10k and 100k LOC) and invested development time (more than several years). The result is a project which allows to approach realistic debugging.

The selected project is JHotDraw.[1] JHotDraw serves as a framework for programs that involve modeling. Its educational background makes it a perfect choice for developing or debugging. This is reflected in the high quality of source code. For demonstrating the usage of the framework, JHotDraw features a few example applications including a drawing application called JavaDraw. JavaDraw is a modeling drawing program. In a regular drawing program, drawing a box results in a collection of pixels which share no longer a reference to a box. In a modeling program the box remains a box in which attributes can be changed such as fill color, boundary color and size. This Object-Oriented property makes it a good test project. Another big advantage of this candidate is the availability of a bug repository, which will be described later. In the more than ten years of development of JHotDraw, multiple versions are released. The choice for the best version is based on the amount of bug reports and size of the source code. The size of the latest version is significant larger than previous versions but contains a fewer amount of reported bugs. The 6b1 version was released before a big architectural change. Based on those properties version 6b1 is chosen. The source code size is 26 KLOC (kilo lines of code) and has the total MCC (McCabe Cyclic Complexity [24]) of 7335 spread over 616 classes. The McCabe metric is based on the features of a directed graph representation of the source code and can be interpreted

[1] `http://www.jhotdraw.org`

as the amount of paths the code can follow. In the Java implementation this also includes object instantiation. After the candidate JHotDraw 6b1 was chosen, the focus is shifted to the bug reports from the bug repository. The bug description from the reports was used to replicate the bugs. The next section selects the bugs for the experiment.

5.1.2 Selecting bugs

With the available bug repository it is easier to use an existing bug than fabricating one. In addition, using existing bugs provide a more realistic solution. JHotDraw has a bug repository in which bugs are reported by the users of the project. The developer who reads the bug report assigns a category and severity. When the bug is fixed a comment is posted back on the report to notify any interested users. These reports are used to determine suitable bugs to evaluate the supporting debugging concepts of Visual Debugger. The supporting debugging consists of supporting locating the cause of the bug. To evaluate this support, the bugs from the bug repository were categorized in two categories by interpreting the bug reports. The first category consists of bugs which are easy to solve: usually by changing one line of code. Most bugs from this category are caused by missing a rare scenario. The second category consists of more complex bugs: these require structural changes to solve. One bug of each category is selected for the experiment. Only this total of two bugs is selected because debugging is very time intensive.

5.2 Setup of the experiment

For an evaluation of the concepts used in the Visual Debugger to answer the research questions, an experiment was setup. The best solution is a controlled experiment [1] to test the variables independently. Unfortunately, such an experiment was not possible. It would require a large number of random selected participants which were not available. In addition, the prototype tool's purpose is to primarily demonstrate the visualization of the neighborhood. The development is still in a beta stage in which certain unintended behavior might occur. Instead, the decision was to conduct a preliminary pre-posttest experiment. This allows formulating an opinion about supporting debugging by comparing the user's expectation and experience with the Visual Debugger. In the pretest the user's expectation is assessed and in the posttest the user's experience. The user's experience is obtained by letting the user using the tool for executing debug tasks between the pre and post test. The results from the debug tasks provide input for answering research question 1 and 3. The pre and post tests consist of questionnaires. The questions are not only about the expectation-experience comparison, but also involve other key points of supporting debugging. In this way the questionnaire's results are used for answering all research questions.

5.2.1 Layout of the experiment

The experiment is divided into components in the following way:

Pre-test questionnaire	5 minutes
Demonstration tool	20 minutes
Executing tasks	60 minutes
Post-test questionnaire	5 minutes
Interview	10 minutes

All parts of the experiment are supported by documentation. The experiment starts with the pretest which consists of a questionnaire. After filling that questionnaire, the functionality of the Visual Debugger is explained by a short demonstration. The demonstration is a convenient way to quickly get familiar with a new tool. In addition, any question the participant may have can be directly answered. The participant is then prepared for the main part of the experiment: executing the tasks. For reference, a short tutorial (see appendix A) of the Visual Debugger is provided. After the tasks the post test which also consists of a questionnaire is given. The last component is a short interview in which any remaining remarks and comments are noted.

5.2.2 Pre- and Posttest Questionnaires

The experiment consists of two sets of questionnaires (see appendix A): the pretest before and the posttest after the debug tasks. The questions are described as statements for which the participants can rate his or her opinion. The rating is based on the Likert scale [23] with a rating ranging from 1 to 5 representing the value “strong disagree” to “strong agree”. The answers provide valuable information for the research questions. The questionnaires are setup with inspiration drawn from other theses [41].

Pretest questionnaire The pretest questionnaire starts with asking the programming experience to determine the qualification of the participants. The target group should be experienced in using the debugger in Eclipse. The following category is to gain more insight into the participant’s debug process. To determine the debug behavior, questions are about their purpose of debugging and usage of debugging functions. The last category of the pretest, after giving a short description of the Visual Debugger, contains questions about the expectation for the debug support of the tool. After the pretest and executing the tasks the posttest questionnaire is presented.

Posttest questionnaire The posttest shares some questions with the pretest questionnaire to research whether the expectation is changed after executing the tasks. In addition, the posttest contains questions concerning the evaluation of the execution of the tasks. This is questioned to answer how the participants perceive the experiment. In particular if the participants could use the Visual Debugger for supporting debugging. The questions on this subject are divided into the categories of tool features and GUI handling. The last category concerns the practicality of the Visual Debugger: would users perceive the Visual Debugger as added value for debugging?

5.2.3 Demonstration

After the pretest a demonstration of the Visual Debugger is given to help prepare the participants with executing the tasks. The tool is demonstrated on a toy test project. The test project contains 5 classes and features object-oriented techniques such as: sub-classing, polymorphism and calls to variables across objects. With this project the source code visualization and other functionality of the Visual Debugger is explained.

5.2.4 Pilot sessions

The next part of the experiment are the tasks. How can the experiment tasks be designed in a way that effectively tests the Visual Debugger concepts? The main idea is locating the cause of the bugs with support of the Visual Debugger. Deciding on the appropriate difficulty of locating the cause is very important to retrieve useful results. Too easy and the usefulness of the Visual Debugger could be neglected, too difficult and the participant could lose interest or get annoyed. After running two pilot sessions and adjusting the parameters of the experiment, the task objectives are optimized. The pilot sessions resulted in multiple modifications: the task description is explained in more detail, additional points that help the participants to start the tasks are added and the time limit of the tasks is determined: 20 minutes for Task 1 and 40 minutes for Task 2. The result is doable tasks in the maximum permitted amount of time of an hour.

5.2.5 Tasks

The tasks description (see appendix A) is constructed from the bug reports and experience with the Visual Debugger. The first part of the tasks consists of locating the bug cause with support of the Visual Debugger. The use of Visual Debugger is encouraged but not mandatory. Using this approach it is possible to see if the participants are able to easily adopt the new tool and benefit from its use as supporting debugging. The second part of the tasks consists of providing a solution to fix the bug. This allows for checking if the participant understood the context of the cause of the bug.

Task 1 Section 5.1.2 mentions that two bugs of different complexity are selected. Each bug is assigned to a task. The first task contains the easier bug and the second task contains the more complex bug. The bug in the first task is the exception that is thrown when running the test program. The purpose of the first task is to find the location of the cause. In this task, the Visual Debugger can support deciding if the cause is located in the particular class of the object that throws the exception or is in another class.

Task 2 The bug in the second task is that the test program's execution shows unexpected behavior when a particular series of user commands is executed. The purpose is to find the location of the cause. This task requires a more in-depth search of dependencies between types and variables. The purpose is to experience the multiple features of the Visual Debugger. In both tasks a breakpoint was provided to help the participant start debugging. The

location of the breakpoint was set in the class which is called after the first user command initiating the bug.

5.2.6 Interview

The last part of the experiment is a 10 minutes short interview. It is conducted with each participant after the posttest questionnaire. Although there is room for remarks and comments in the posttest questionnaire, the interview provides new interesting information. In general the interview questions concern the experience with the experiment and the Visual Debugger, in particular the positive and less positive points, for example:

- What is your overall thought about the experiment?
- Did you encountered any problems during the experiment?
- What features of the Visual Debugger were useful?
- What parts of the Visual Debugger could be improved?

5.2.7 Environment

The conditions in which the experiment is conducted are attempted to keep equal for all participants. The screen space of the monitor is an important requirement: all computers used for the experiment had a 24" monitor with a 1920x1200 resolution. The software environment used for the experiment consisted of the Visual Debugger plug-in integrated in the IDE Eclipse version 3.6 and was deployed on 3 different platforms: MacOS, Windows and Linux.

5.3 Summary

This chapter explains the details of the experiment setup. The experiment starts with the preparation in which the project JHotDraw is chosen. The setup of the experiment consists of the pretest, demonstration, tasks, posttest and interview. The pretest asks questions concerning the participant's experience in developing, debugging and tool expectation. The tasks contain locating the cause of one simple and one complex bug. The posttest questionnaire questions about the task experience, tool usage and tool practicality. The interview consists of questions concerning the experience of the experiment and the Visual Debugger usage.

Chapter 6

Experiment results

The experiment provides different kinds of results: quantitative ratings from the questionnaires and textual answers from the tasks. The results are analyzed in the following order: first the development and debugger experience, followed by the tasks, followed by the tool evaluation. Each part is further divided per category. The task results also contain the results from the breakpoint usage. After the tool evaluation, the analyses from the results are discussed.

6.1 Development and debugger experience

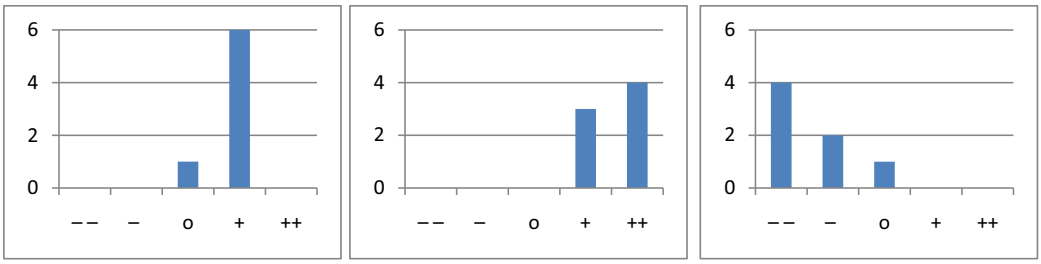
The experiment results start with the development and debugger experience which was acquired from the pretest questionnaire. This information is used to sketch the debug behavior of the participants.

6.1.1 Development experience

The results from the experiment start with the assessment of the participants. For the participants to provide valuable information, a few requirements should be met. The first requirement is that the participants consider themselves experienced in software developing. Additional requirements are familiarity with Eclipse and unfamiliarity with JHotDraw. To assess the program understanding support of the Visual Debugger, it is required to not know about the internal working of JHotDraw. Seven participants participated in the experiment. The results from the questionnaires and tasks are transformed in more easily interpreted diagrams. The charts in Figure 6.1 show that almost all participants consider themselves experienced Java developers, often use Eclipse and are unfamiliar with the test project JHotDraw.

6.1.2 Debugger experience

The next category of the questionnaire focuses on the debugger experience of participants. One of the key aspects of solving a debug problem is program understanding. For supporting debugging we want to know if the debugger is used for that purpose. The question is

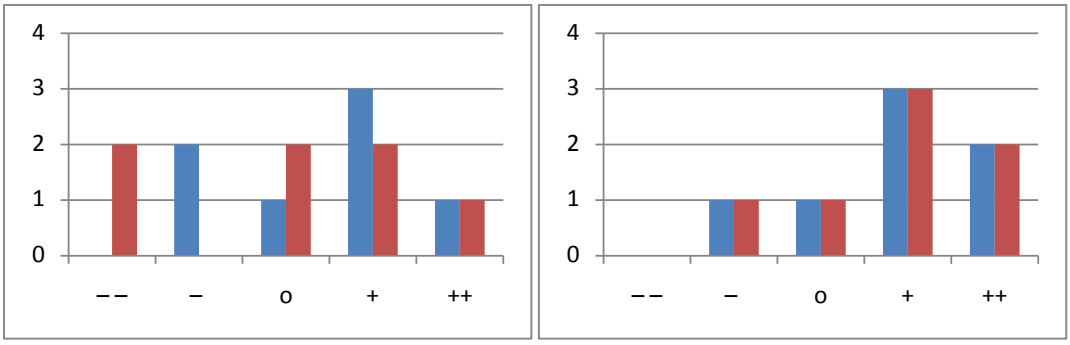


(a) Developing java (b) Often use Eclipse (c) Familiarity with JHotdraw

Figure 6.1: Histogram of participants developing experience. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

for what purpose the (Eclipse) debugger is used often: program understanding and/or debugging. Chart (a) in Figure 6.2 shows that the participants use the debugger slightly more for debugging (blue columns) than for program understanding (red columns). But the small difference indicates that program understanding is also a big part.

Debug methods How do the participants debug? Are there other methods to perform debugging besides the debugger itself? The next question is if the Eclipse debugger is the main debug option or if there are other methods used. Chart (b) in Figure 6.2 shows that besides the Eclipse debugger (blue columns), other methods (red columns) such as logging and 'println' are often used. Besides the Eclipse debugger, participants mentioned other debug tools. One other tool mentioned is git bisect.[1] Git bisect outputs the differences between revisions of source code. By marking revisions good or bad, the first revision containing the bug can be found. This marking is accomplished by writing a test case to determine the presence or absence of the bug.



(a) Use for pu or debugging (b) Preferred : Eclipse debugger or other

Figure 6.2: Histogram of participant's debugger experience. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

[1] http://www.kernel.org/pub/software/scm/git/docs/git-bisect.html

Starting points The next category continues with the question how participants debug. Debugging consists of understanding the program's behavior. When inspection of the source code is not sufficient to reveal this information, the program's execution is inspected. Because this inspection is accomplished at runtime, the focus is on the actions when the debugger is suspended. The main method for suspending the debugger is using breakpoints. Therefore the research concerns the starting points for breakpoints. The Visual Debugger is based on breakpoints. To decide where to place the breakpoints, the question is which starting points for debugging are used. Four options are available as answer. The first option is test case, in which a part of the program's input and output is isolated to test a specific scenario. The second option is action handlers which consist of actions such as thrown exceptions or certain method calls. The third starting point option is the main method which indicates that the program's execution is followed from its beginning. The fourth and last option is stack traces which are available when exceptions are thrown and the debugger is suspended. Figure 6.3 shows that participants mostly use stack traces and action handlers are used least. The main method and test cases options receive both a more neutral rating.

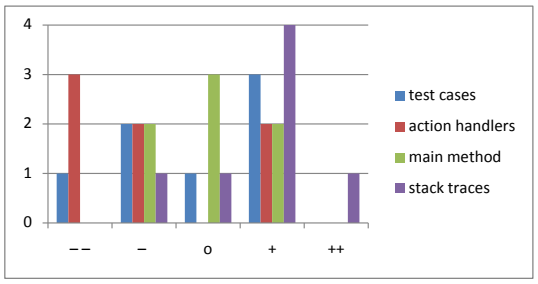
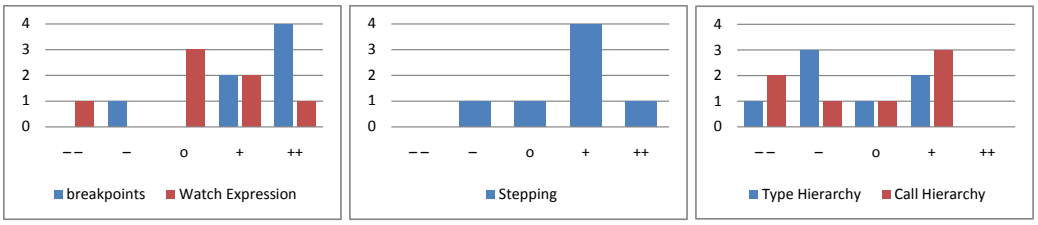


Figure 6.3: Histogram of usage of starting points. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

Inspect objects To determine the debugger behavior of the participants, the next question is about the usage frequency of different debugger functionality. In particular the functionality with the purpose to inspect the state of the objects at runtime. There are two options: watchpoints and breakpoints. Watchpoints are based on values from fields of objects and breakpoints are based on source code locations such as method calls and exceptions.

Chart (a) in Figure 6.4 shows that participants use breakpoints more than watchpoints to inspect objects. The more general usability of breakpoints are rated higher than the more specialized watchpoint functionality.

Stepping usage The next functionality questioned is the execution's control flow. The debugger stepping functionality provides the ability to step through single lines of code. This allows following the program's control flow closely but it is also difficult to keep track of the more global behavior. Is the stepping functionality used often? Chart (b) in Figure 6.4 shows that participants often use the stepping functionality.



(a) Object inspect method (b) Often use stepping (c) Dependency inspect method

Figure 6.4: Histogram of participant's use of debug features. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

Dependency search The last researched debugger functionality concerns the retrieval of the dependencies. Not all functionality for supporting debugging is found in the debugger itself. For providing an overview of the source code dependencies, Eclipse has the functionality to show the types hierarchy and the methods hierarchy. Are dependency searches mostly based on types or methods? The options in the answers are: often use of the 'Type hierarchy' and/or the 'Call Hierarchy' feature of Eclipse. Chart (c) in Figure 6.4 shows that participants use the method based (Call Hierarchy) search more often.

From the development and debugger experience results the next part is the results from the tasks.

6.2 Tasks results

Between the questionnaires lies the main component of the experiment: the tasks. The two tasks are described and evaluated. Also the usage of breakpoints for the debug problems is evaluated.

6.2.1 Task assignment

A debug problem can be divided into several components: determine the kind of bug, locate the cause of the bug and fix the bug. Locating the cause of the bug with debug information is no easy task (for example see [11]). The capability of supporting locating the cause of the bug is tested for the Visual Debugger. This is accomplished by showing the neighborhood surrounding the user's set breakpoints. The visualization of the neighborhood allows to determine the dependencies that might be responsible for the bug. This will support finding the cause of the bug. The purpose of the tasks is to verify or reject that this approach supports debugging and adds value to existing debuggers. An additional question is if this process can be applied to a complex causality chain of the bugs. Therefore bugs with different complexity are tested.

For the results of the tasks, the task documentation contains a blank section in which the participants could write down the insight gained using the Visual Debugger. For each task

in the participants were asked to a) locate the cause of the bug and b) provide a possible solution.

6.2.2 Task 1

The first task consists of finding the cause of a particular runtime exception in the source code. It was thrown after a menubar command (see appendix A) was issued by the user.

The cause of the exception lies in a variable 'fe' that should be initialized. Somewhere else in the class this variable is also accessed but preceded by a null check. A fix for the bug is easily done by also adding a null check at the location where the exception is thrown (see Listing 6.1).

```
1 public void alignAffectedFigures(Alignment applyAlignment) {
2     FigureEnumeration fe = getAffectedFigures();
3     if (fe.hasNextFigure()) {
4         Figure anchorFigure = fe.nextFigure();
5         ...
6     }
7 }
```

Listing 6.1: Fix for Task 1.

All participants were able to come up with this answer (see Table 6.1). One participant even investigated the bug further and mentioned that the particular menu command should not even be enabled yet (the command enabling should be preceded by creating a model object). Two participants mentioned that this task was easy and thought that using the Visual Debugger would actually slow down debugging and reverted to using conventional methods.

Table 6.1: Tasks results. The numbers indicate the amount of participants.

	Task 1	Task 2
Able to locate cause	7/7	5/7
Provided a solution	7/7	2/7

6.2.3 Task 2

The second task is to locate the cause of a logical design error in the source code. The JHotDraw's drawing program provides the ability to combine two objects into one so called composite object. The attributes on that object such as color and width are applied to the objects which the composite object consists of. Also, JHotDraw contains an undo command which undoes the most recent executed command. In the task, the series of commands is as follows: create a composite object of two objects, use the fill color command and finish with the undo command. The last command should undo the fill color command and should

change the fill color of the composite object to its original fill color, but this does not occur. The task is to find out why the undo command does not work as intended. Finding the cause is not easy: the sequence of commands involving different suspects (color, commands, figures) makes this task a complex task. The answer should include the location of the cause of the bug and a short explanation why this logical error occurs.

The reason why the undo command does not work in this situation is a design flaw. Figure 6.5 shows an overview of the key classes. The fill color command is a command that changes an attribute of a figure. This command can be applied to a selection of multiple figures. The change attribute commands, such as the fill color command, store the attribute value before the change, which allows the undo command to undo. The fill color command applied on the *composite* figure will store the original value of the fill color. But because this color is not set (what is the color of two different figures?) the original value is not stored and the fill color cannot be undone. A fix for this problem is difficult and requires structural changes in the source code. A solution is that change attribute commands map each (composite) figure with multiple original values. Another solution is that the figure classes have the responsibility of the 'original values' and not the change attribute commands. This last solution is actually applied in the current version 7 of JHotDraw.

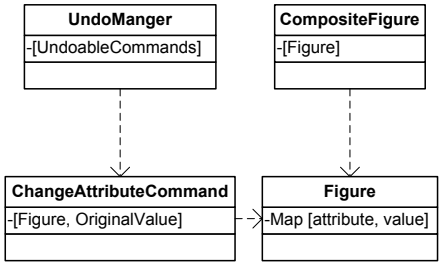


Figure 6.5: Key classes for Task 2. The arrows indicate dependencies. The attributes in the classes surrounded by '[...]' are a Collection type.

```
1  public void setAffectedFigures(FigureEnumeration fe) {
2      ...
3      // then get new FigureEnumeration of copy to save attributes
4      FigureEnumeration copyFe = getAffectedFigures();
5      while (copyFe.hasNextFigure()) {
6          Figure f = copyFe.nextFigure();
7          Object attributeValue = f.getAttribute(getAttribute());
8          if (attributeValue != null) {
9              addOriginalValue(f, attributeValue);
10         }
11     }
12 }
```

Listing 6.2: Location of the cause of the bug in Task 2 in ChangeAttributeCommand.java.

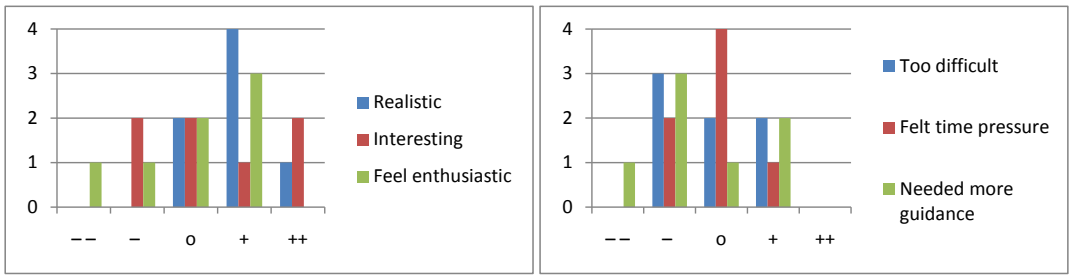
Table 6.1 shows that five participants were able to locate the cause of the bug. The criterium for this is mentioning the location where the original values are stored (see Listing 6.2). Two participants were able to deduce a solution that consisted of storing multiple original values per composite figure.

6.2.4 Tasks evaluation

This section evaluates the tasks. Are the tasks representable as a realistic task and complex enough to demonstrate the potential benefit of the Visual Debugger? This category consists of both positive and negative formulated questions.

Positive points The first question of the positive points is if the tasks are realistic (blue columns in chart (a) of Figure 6.6) as debugging tasks. Most participants agreed. The next question is if the tasks are interesting to do (red columns in chart (a) of Figure 6.6). If participants were reluctant to complete the tasks then this could negatively influence their effort. The results show a positive response. The next question is if the participants were enthusiastic about the tasks (green columns in chart (a) of Figure 6.6). The response was neutral.

Negative points The next set of questions contains the negative points. The first question of the negative points is if the tasks are too difficult (blue columns in chart (b) of Figure 6.6). The reason was not specified and could range from not understanding the test program, purpose of the tool or task to problems with interpreting the tool's visualization. The answers were balanced around neutral. The next question is if the participants felt time pressure (red columns in chart (b) of Figure 6.6). The response rating was almost neutral. The last question in this section is if the participants preferred more guidance in completing the tasks (green columns in chart (b) of Figure 6.6). A big part of the guidance was provided by the demonstration. The majority of the participants would like some more guidance.



(a) Positive points for task assessment (b) Negative points for task assessment

Figure 6.6: Histogram of participant's task assessments. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

6.2.5 Breakpoint use

In the debug process there are several ways to obtain runtime information to investigate the program's behavior. For a close inspection of the program's state, the execution is interrupted with the use of breakpoints. Breakpoints are user marks of interesting points in the source code to find out what is happening [21]. For the tasks in this experiment, the participants were asked to store their breakpoints. Did the breakpoint usage differ among participants? This question is asked to gain insight into the usage of breakpoints. Because the usage of breakpoints was researched after the first participant has taken the experiment, one participant is removed from the results.

The approach is as follows: assume a situation in which a program with a bug is run. At some point a particular method is called. A few code lines later, the code line which contains the bug is executed but its result is not immediately visible to the user. The bug reveals itself at some more code lines later by resulting in unintended behavior of the program. Lets define the starting point as a method which precedes the cause of the bug. The choice is arbitrary but should be close to the cause to eliminate the search area for the cause. The end point is defined as the method in which the bug is revealed (the result of the bug is visible to the user). Lets define the bug path as the method calls between the start and end point. When debugging a program, the breakpoints should lie within the bug path. For the breakpoint usage of the participants we looked at the breakpoint distribution of the bug path.

Task 1 The starting point in Task 1 is the given breakpoint (located in the Class CommandMenu) in the task description. Between this starting point and the end point (for Task 1 the thrown exception) there are 100 method calls (excluding calls to the Java API and including return calls). From these calls there are 35 unique calls, which means there are 35 possible locations for the cause of the bug. Table 6.2 shows a small usage of breakpoints (1.8 on average). We assume stepping is used more. In addition, in unfamiliar code the participant needs to discover the interesting parts of code. Table 6.2 also shows that the

Table 6.2: Breakpoint distribution for Task 1. The numbers indicate the amount of participants using a breakpoint in that method.

Type name	Method name	Participants
AlignCommand$UndoActivity	alignAffectedFigures	5
AlignCommand	execute	1
FigureEnumerator	nextFigure	1
UndoableCommand	execute	2
CommandMenu	addMenuItem	1
CommandMenu	enable	1

participants agreed on which methods contain interesting parts of code.

Task 2 For Task 2 the bug did not provide a hint of the location of the cause in the source code. In the task description it is suggested to start from the last undo command. This bug path consists of 69 method calls divided over 22 different methods. The bug path only reveals that the cause of the undo command is dependent on the preceding command. To find out how, more debugging is required. The second bug path starts from the preceding command which is the fill color command. This consists of 73 method calls divided over 29 different methods. When combining the two bug paths, it results in 46 different methods. Table 6.3 shows that participants use more breakpoints (3.3 on average) divided over more methods in comparison with the breakpoint distribution of Task 1. The breakpoint use is concentrated on the undo and the execute methods.

Table 6.3: Breakpoint distribution for Task 2. The numbers indicate the amount of participants using a breakpoint in that method.

Type name	Method name	Participants
ChangeAttributeCommand$UndoActivity	undo	3
ChangeAttributeCommand$UndoActivity	redo	1
ChangeAttributeCommand$UndoActivity	addOriginalValue	2
ChangeAttributeCommand$UndoActivity	getOriginalValue	1
ChangeAttributeCommand$UndoActivity	setAffectedFigures	2
ChangeAttributeCommand$UndoActivity	UndoActivity	1
ChangeAttributeCommand	execute	3
UndoCommand	execute	3
UndoableCommand	execute	1
UndoableAdapter	getAffectedFigures	1
UndoManager	popUndo	1
DecoratorFigure	setAttribute	1

With the task results evaluated, the next section describes the evaluation of the Visual Debugger tool.

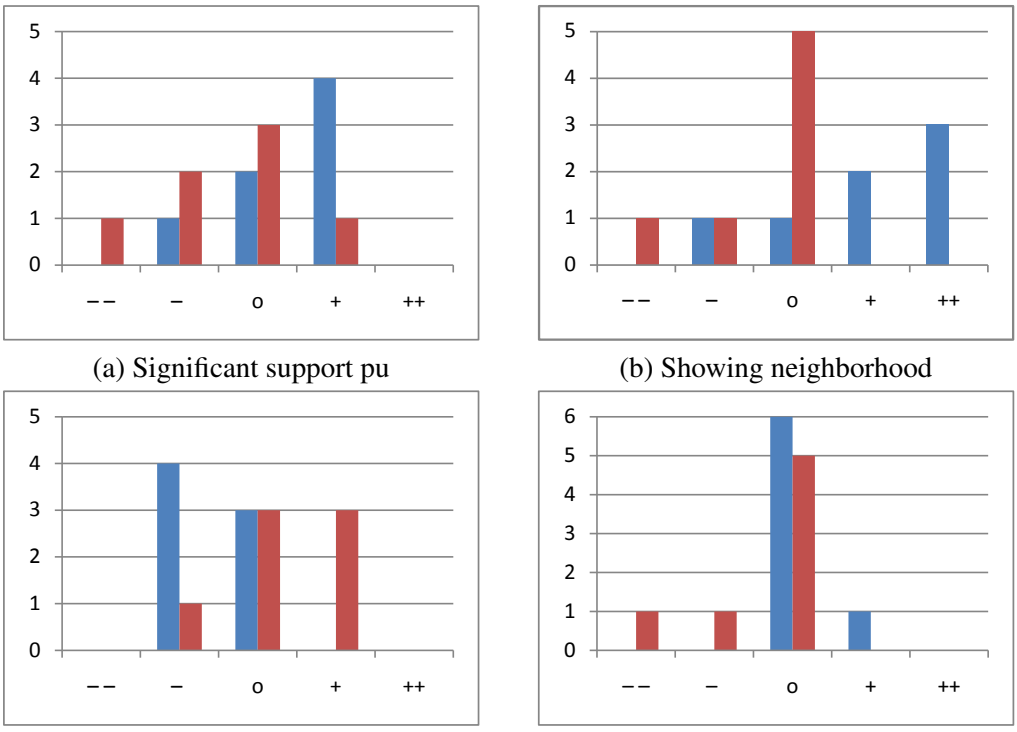
6.3 Tool Evaluation

This section divides the evaluation of the tool into the categories: tool features, GUI handling and practical use. For supporting program understanding and debugging there is a comparison made in: the expectation before executing the tasks and the perceived experience after executing the tasks. Additional feedback is also provided from the participants sharing their view on using the Visual Debugger in the experiment.

6.3.1 Comparison of expectation and experience

This section compares the expectation and experience of the Visual Debugger usage. Has the experience with the Visual Debugger changed the participant's expectation about the tool? The comparison is accomplished by asking the same set of questions concerning the supporting debugging of the tool in the pretest questionnaire and posttest questionnaire. Because the tasks did take quite some time, the answers should not suffer from a repeating answer pattern. The first category is program understanding.

Support program understanding The first question is if the tool could significantly support program understanding (chart (a) in Figure 6.7). Prior the experience the expectation (blue columns) was rated mostly positive. After the experience (red columns) this enthusiasm dismissed a bit to neutral. The second question is if showing the neighborhood for the executing point would benefit program understanding (chart (b) in Figure 6.7). Before the tasks the expectation was very high but after executing the tasks the opinion was reduced to a neutral position. The third question is if the additional user interaction (for tool usage) would hinder program understanding (chart (c) in Figure 6.7). It is possible that it is too distracting. Prior executing the tasks the participants would expect no problems. Executing the



(a) Significant support pu

(b) Showing neighborhood

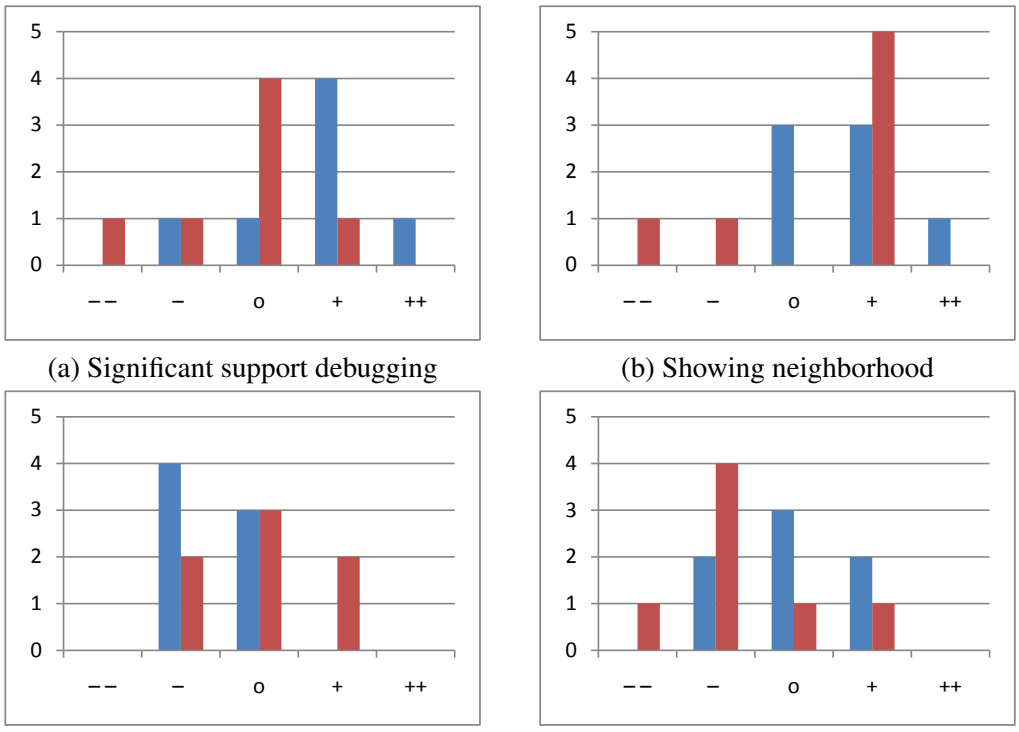
(c) User interaction hinder pu

(d) Use extensively for pu

Figure 6.7: Histogram of comparison of program understanding (pu). Blue columns present the pretest and red the posttest results. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

tasks did provide insight and changed their opinion to a more negative stand (the question was asked in a negative formulation, so higher rating means more hinder). The fourth question is if the participant would use the tool extensively for program understanding (chart (d) in Figure 6.7). Before executing the tasks the opinion was neutral and after the tasks it mostly remained unchanged.

Support debugging The second subject of the comparison is debugging in which the same questions as the program understanding category were asked but for the purpose of supporting debugging. The first question is if the Visual Debugger significantly supports debugging (chart (a) in Figure 6.8). The participants agreed with the question in the pretest. In the posttest the total rating was declined to neutral. The second question is about showing the neighborhood of the executing point (chart (b) in Figure 6.8). Most participants were neutral and positive, in the posttest almost all participants were positive. The third question is about hinder of user interactions (chart (c) in Figure 6.8). In the pretest the results show a mostly negative opinion. In the posttest the rating changed to a mostly neutral position. In the fourth question the extensive use of the Visual Debugger for debugging is questioned (chart (d) in Figure 6.8). The pretest results show a mostly neutral opinion and is changed



(a) Significant support debugging

(b) Showing neighborhood

(c) User interaction hinder debugging

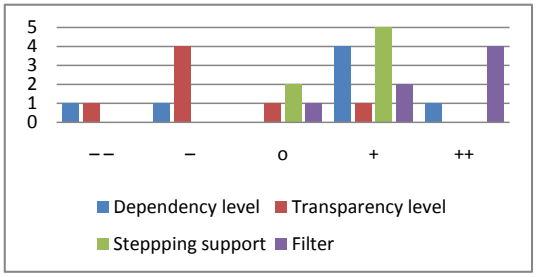
(d) Use extensively for debugging

Figure 6.8: Histogram of comparison of supporting debugging. Blue columns present the pretest and red the posttest results. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

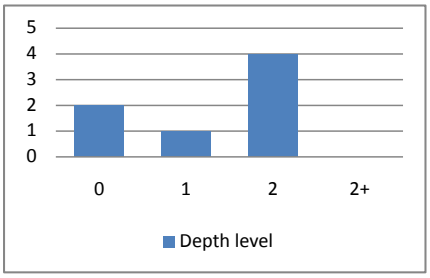
in the posttest to a more negative opinion.

6.3.2 Tool features evaluation

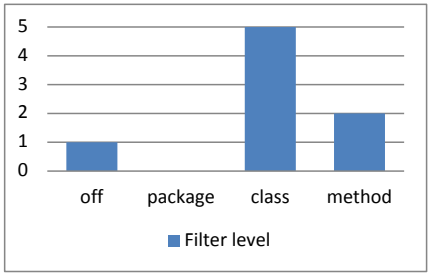
This section covers the favorite user settings and applies a rating to the usefulness of the Visual Debugger's concepts. The first feature of the tool that is covered is the dependency setting. This setting controls the neighborhood size. The blue columns in chart (a) of Figure 6.9 show that this setting is highly appreciated. The most used setting is 2 (see chart (b) of Figure 6.9). The next feature of the tool is the transparency setting. This is not unsurprisingly rated lower (red columns in chart (a) of Figure 6.9) because this setting is a GUI preference setting. The next feature is stepping support. This is a fundamental functionality of the Visual Debugger. Almost all participants were positive (green columns in chart (a) of Figure 6.9). The last feature is the filter setting, which is the other most important supporting debugging feature. The participants' response is mostly strongly agreeing (purple columns in chart (a) of Figure 6.9). Most participants used the "class" filter setting (chart (c) of Figure 6.9).



(a) Usefulness of customized features



(b) Preferred dependency setting



(c) Preferred transparency setting

Figure 6.9: Histogram of features usefulness and usage assessment. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

6.3.3 Gui handling evaluation

This section covers the perceived GUI handling experience. The first statement is that the Visual Debugger requires much user navigation such as zooming and panning (blue columns

in Figure 6.10). The tool available settings should give the user enough options to reduce the total amount of visible elements but could also require more user intervention. Most users strongly agreed that they navigated often. The next statement is if the function of filtering and adding nodes or dependencies worked as expected (red columns in Figure 6.10). If this is not the case it could be that the function is confusing and therefore less used. Most users agreed that the function worked as expected. The next statement is if the users used the filtering and adding often (green columns in Figure 6.10). Despite understanding the working of the feature most users disagreed using it often. This could be that there was no need or maybe they thought it would break the correct representation of the source code. The last statement of this category is if the screen space is sufficient (purple columns in Figure 6.10). This statement is related to the amount of graph elements and zooming setting. Most users disagreed and would like more screen space.

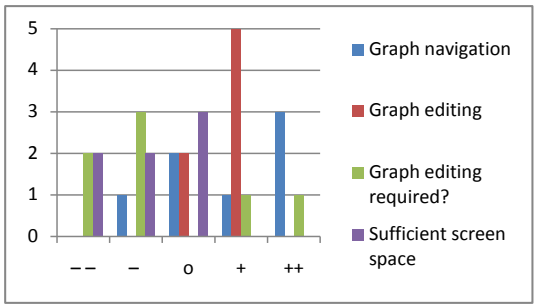


Figure 6.10: Histogram of GUI handling assessment. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

6.3.4 Perceived practical use

The final section of statements in the posttest questionnaire contains the questions concerning practicality. Would the users want to use the Visual Debugger for professional debugging? The first statement is if the graphical presentation of source code provides added value for debugging (blue columns in Figure 6.11). Most participants agreed. The next statement is if the graph manipulation is intuitive (red columns in Figure 6.11). Most participants agreed but a few disagreed. The next statement is if the tool is easy to use (green columns in Figure 6.11). The result is the same as the previous statement: most participants agreed, a few disagreed. The next statement involves if the tool is ready for daily use (purple columns in Figure 6.11). Most participants disagreed. The last statement is the most important one: if the tool is useful as extension for the debugger (turquoise columns in Figure 6.11). Most users agreed, only one participant disagreed.

6.3.5 Comments from posttest and interview

The last question of the posttest is if the participants had any remarks left to share. In addition, after the posttest questionnaire was filled, the interview also provided some feed-

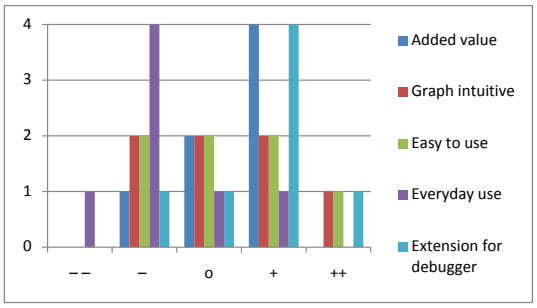


Figure 6.11: Histogram of practical use assessment. Horizontal axes: rating ranging from strongly disagree (—) to strongly agree (++). Vertical axes: number of participants.

back. A part of the comments are initial concerns with adapting to the new tool. One such comment is that the initial graph contains too much information that may confuse the user. It is possible that the user may have overlooked or forgotten the option to start the Visual Debugger by only selecting the interesting packages. Other concerns were mentioned:

- Link between graph and debugging not immediately clear
- The tool is slow
- See its purpose in program understanding but less in debugging

Some participants were confused about its purpose: 'could not understand the support in debugging' and assuming it will find answers for you. In the category of graph navigation some participants found they had to zoom too much, others found the graph easy to understand and navigate. Some disliked that the mouse scrolling setting was set on zooming instead of scrolling. Besides these concerns there are also positive comments:

- The tool is especially helpful for providing an overview
- Has potential for new users

Other comments contain recommendations for future features. The first mentioned feature is highlighting the call chain. Currently only the method where the execution is suspended, is highlighted. Highlighting preceding executed methods would introduce a visual history of method traversal. This feature lies more in the direction of tracing. Other features include:

- Roaming ability in the visualization
- Double click on a node to view the source code
- Move settings to the debug perspective
- Add visual difference between dependency levels
- Ability to add breakpoints using the graph representation of methods

Other features include additional feedback: 'warning before showing many elements' and 'loading indicator'. The most requested feature is support of dynamic dependencies in addition to the support of static dependencies, for example showing the dependency between calls to different subclasses at runtime.

6.4 Discussion

The results in the experiment are presented and analyzed. This section uses those results to answer the research questions.

6.4.1 Debugger use

For research question 2: *What are key points for supporting debugging?* information is gathered about the debugger use. By asking how the debugger is used, key points can be identified for supporting debugging.

The debugger use is questioned in the pretest questionnaire. The results show that the (Eclipse) debugger itself is not solely used for debugging and not necessary the main debug method. What remains unknown is if this is related to the unfamiliarity of the source code. Getting insight into the (runtime) behavior of the program requires program understanding, which is one of the recognized purposes of the debugger. When looking at the use of debugger features, the debugger is used for close inspection: the stepping functionality, breakpoints and watch expression features are often used. Other favorable sources of information for setting breakpoints include stack traces and test cases. Program understanding features of the IDE are also used for debugging. When searching for dependencies Eclipse provides options to search dependencies based on methods and types. Dependencies for methods are more often searched for than types.

6.4.2 Task results

Research question 1: *Can local neighborhood visualization be used for supporting debugging?* is answered by looking at the results from the tasks. The practical experience combined with the questions about the Visual Debugger's use, provides a good indicator whether the concept of local neighborhood visualization can be used for supporting debugging.

From the results of Task 1 it is apparent that the Visual Debugger cannot be effectively applied to all bug cases. For Task 1 the fix could be applied in the same method in which the bug revealed itself. The inner workings of methods are not visualized with this tool. But what can be visualized are the dependencies of the variables. In this case, the absence of dependencies to other classes could indicate that the problem was more local. Task 2 uses a scenario that contains a more complex bug in comparison with Task 1. The results show that with the support of the Visual Debugger, also the cause of more complex bugs could be located. The effect of program understanding is less pronounced: not all participants were

able to provide a fix for the complex bug. More research is required to point out when the support is beneficial and how it is effectively used.

From the posttest questionnaire, the participant's experience with the tool in the tasks and the tool's functionality is evaluated. The first category is task evaluation. For the most part the response was answered neutral. This is not a bad result: fine-tuning the task requires dealing with many parameters and using a new tool takes time to adjust. The features of the Visual Debugger were also individually evaluated. The tested features are all rated positively as useful. The only feature rated lower is the transparency setting which is less important for debugging. Besides presenting software in the graph in an optimized way, the GUI handling itself is separately evaluated. Overall the GUI handling of the tool could use some improvement for future work. New smart ways to present information in limited screen space should decrease the need for the navigation in the graph. Another option is to add more (advanced) filtering-and-adding-options of graph elements. In the last category of the posttest questionnaire all aspects are considered to assess the practical use. Overall it was perceived fairly positive especially for 'added value for debugging' and for 'extension for the debugger', although for daily use some improvement is required.

6.4.3 Comparison of the user's expectation and perceived experience

Research question 3: *Is the use of debugger for program understanding (general) and debugging (specific)?* is answered by looking at the affect of the tasks on the user's opinion about the program understanding use and the debugger use of the Visual Debugger.

The impact of the tool was apparent in the comparison between the expectation and experience with the tool. In the first series of questions it concerned the usability for program understanding. In overall, the results from the pretest show a small positive opinion that changed to neutral in the posttest. The role of program understanding for the Visual Debugger should be better defined to be better noticeable. For the Visual Debugger's debugging support the experience went in a slightly positive direction. From all the questions, the most remarkable result is the showing the neighborhood feature which was first rated neutral but after experience with the tool changed to a mostly positive rating. Showing the neighborhood has a supporting debugging role because it provides context. The dependencies would else be hard to discover through other debugger functionality.

6.5 Threats to validity

Experiments must be careful constructed to only research isolated aspects and exclude other aspects. This is accomplished by looking at the cause-effect inferences made during the analysis and the generalizability of the results: the internal validity resp. external validity.

6.5.1 Internal validity

One influence on the results was that the participants were inclined to answer with a positive bias. This was countered by mentioning that the value in the research is in rather true than 'good' results. Also the documentation urged to answer trueful. A second influence could be time pressure. Under pressure the results might be less precise. Although the tasks in the experiment state a maximum time limit, for the results it was more important to obtain experience with the Visual Debugger. The participants were instructed to write down the insights they learned within the given time. A final point of inference is a steep learning curve. Being unfamiliar with the Visual Debugger requires some time to adjust the 'normal' debug behavior. Experienced developers have worked with many developing tools and are able to quickly adapt to new tools.

6.5.2 External validity

This section covers the external validity. The first point is that the set of participants is not divers enough, it ranges from MSc students to Postdocs. But the most important aspect is the development experience which is quite divers in background. The second point is the possibility of an unrepresentative test project. When the scale does not measure up to industrial programs it could not be used. But the amount of complexity and KLOC shows otherwise. A final point is unrealistic tasks. This was also questioned in the posttest. But as the tasks include solving an existing bug, it is representative as a realistic case.

6.6 Summary

This section describes the results of the experiment. The results are collected from the questionnaires, tasks and interviews.

The results from the pretest questionnaire show that participants use debuggers for, besides debugging, also for program understanding. In addition, it shows that debugging is not performed by only a debugger but also by other methods. For setting starting points, participants mostly use stack traces. For inspecting objects, participants use breakpoints and stepping functionality, the 'Call hierarchy' for finding call dependencies.

The results from the tasks show that in Task 1 not all participants were convinced that the tool can be used for supporting debugging. The more complex Task 2 presented a more difficult challenge in which more program understanding was required. Two participants were able to use the tool effectively for supporting debugging and program understanding but some participants would require more learning time to adapt their debugging behavior. In the task evaluation most participants rated the tasks very positive although some asked for more guidance.

The results of the breakpoint usage show that the participants set their breakpoints in similar methods when trying to solve the tasks. For the more complex Task 2 the distribution of breakpoints in methods was more spread.

In the comparison between expectation and experience with the tool the questions were divided into two categories: support of program understanding and debugging. In the program understanding category the overall result was that the expectation was rated positive and the experience was more neutral. In the debugging category the 'showing neighborhood' to support debugging was rated higher in the experience.

The posttest questionnaire evaluated the participant's tool experience. The tool feature evaluation shows that the filter feature was rated highest in usefulness, the second highest rating was for the stepping support. For preference the participants preferred a depth level of 2 of the dependence setting and the 'class' setting of the filter setting. The GUI handling evaluation results show that participants on one hand needed much navigation of the graph and on the other hand were reluctant to use the filter/adding functionality despite stating that the functionality's purpose was clear. For practical use of the tool, participants found that the Visual Debugger has added value for debugging and is useful as extension for the debugger.

The experiment also provided room for feedback from the participants. Most concerns were about the learning curve and purpose of the Visual Debugger. Also positive reactions were mentioned: some participants indicate they saw support of debugging by providing overview. Other comments suggested improvements for future releases such as support for dynamic (runtime) dependencies.

With all those results a good base could be constructed to answer the research questions. For research question 1 the tasks and posttest questionnaire provided more insight: supporting debugging was noticed but it required some learning for effective use. For research question 2 the pretest questionnaire provided more insight: debugging is more than the debugger use. For research question 3 the posttest questionnaire provided more insight: the program understanding part was less clear in comparison with the debugging part.

Also the threat validity of the experiment was described. Internal threats consist of positive bias, time pressure and learning curve. The external threats involve participant's diversity, unrealistic test project and unrealistic tasks.

Chapter 7

Related work

This chapter describes related work that have similarities with the supporting debugging concepts of the Visual Debugger. These concepts are usually demonstrated with tools. The tools are divided into the categories: visual debug, debug and program understanding. The most important features of the tools are described and compared with the Visual Debugger.

7.1 Visual Debugging

The first category is approaches from the domain of visual debugging. These are divided into two categories: source code and non-source code based visualization. Tools are considered source code visualization when parts of the source code of the debugged program is represented in a visualization to support debugging.

7.1.1 Non-source code based visualization

To provide easy access to debug commands, a graphical user interface is required. Zeller et al. has build a debugger front-end [42],[43]. The debug tool DDD is a GUI to provide faster access to the many debug commands of GDB.[1] DDD provides the ability to visualize data values. This is achieved by selecting source code lines where the data is referenced. The visualization is updated when GDB is stopped by breakpoints. From the visualization, values can be edited. Also data structures such as linked list can be visualized: the representation of a linked list is a graph containing a cycle. This feature is accomplished by recognizing identical memory addresses.

Moher has build Provide, a visualization and debugging environment for a subset of C [27]. It provides the ability to visualize data values. Another noticeable feature is: patching of source code. Patching allows to execute modified source code while debugging the program. Another supporting debugging feature is the ability to store checkpoints by storing state changes. Provide uses this feature to update modified data values by restarting a stored checkpoint. The values can be directly modified in the bar charts visualization. The bar

[1] `http://www.gnu.org/software/gdb/`

chart is constructed from template visualizations that are mapped (by a user) with program variables.

Ko uses slicing in its supporting debugging tool Whyline [16],[17]. With the tool, the user can specify a question to indicate a particular behavior. Whyline compares this with the program's output and shows a causality of assumptions that leads to the answer of the question. Such question is specified in the following way: Why did text `a` contains value `c`? Whyline provides a typical answer: boolean `x` is `true` after variable `y` is set `2`. This approach allows to verify the assumptions made by the user of the causality path. The tool provides the ability to show for each assumption the responsible code and visualizes the sequential order of the assumptions.

The visual debug tools described all use a visualization based on (object) values extracted at runtime. The Visual Debugger does not provide these values because they are already provided by the Eclipse debugger. The approach of Whyline is an interesting idea but is limited to the type of questions that can be asked.

7.1.2 Source code based visualization

One source code based visualization is the commercial jBixbe,[2] build for debugging purposes. The visualization consists of class diagrams and sequence diagrams. The dynamic runtime information can be inspected by selecting objects from a list. Its visualization presents object dependencies and sequence diagrams.

Another approach of a visual representation of software is Debugger Canvas.[3] Its main concept is code bubbles for inspecting the debugging code. Code bubbles are text fields containing a method of the source code. The code bubbles can be created from calls present in the stack frame of the debugger. The call dependencies option creates new bubbles which are connected with lines. The result is a graph with source code as nodes.

A trace debug tool from De Pauw et al. is Jinsight [30]. The trace is user specified through an iteration process. In this iteration process each iteration should increase the specification of the trace. The trace is defined by selecting methods and associated attributes. Adjusting the specification does not require the restart of the system because the adjusting is performed simultaneously with the debugging program. To support the specification, Jinsight provides a visualization based on the sequence diagram. The diagram is extended with type information by coloring the life lines. Another extension in the visualization is the available abstractions: classes and threads.

Systä et al. build a reverse engineering environment supporting program understanding [36]. The environment is composed of Rigi[4] and SCED. The approach consists of combin-

[2] `http://www.jbixbe.com`

[3] `http://research.microsoft.com/en-us/projects/debuggercanvas/`

[4] `http://www.rigi.csc.uvic.ca`

ing static and dynamic analysis. SCED constructs sequence diagrams which show interaction among objects. The sequence diagrams are build from information extracted from a custom debugger. The state chart diagram shows the behavior of one object. State diagrams are constructed from one or more sequence diagrams. The Rigi views show the relationships between objects such as calls and inheritance. A selection within the Rigi view can provide input for SCED tool or the other way around. This way of linking tools provides the ability to filter uninteresting parts. For additional analysis of software components, support is provided for calculating metrics such as complexity, coupling and cohesion.

The source code based visualization all have in common the hierarchical view. Class, state and sequence diagrams will all work for partial code view, but can quickly meet their limits in screen space when inspecting large programs. In addition, much user interaction is required to specify the interesting parts of source code or objects. The Visual Debugger's abstraction and filter concepts in combination with the automatic navigation concept provides a more automated user interface for supporting debugging.

7.2 Debugging

This section describes debugging techniques without visualization.

The dependency-based model to support debugging in the Visual Debugger has at highest level of detail methods and fields. The bug has to eventually be found by inspecting selected code. A step further in supporting debugging is inspection at a higher level of detail: at the statement level. Assignments of values to objects and references are the basis in the value-based model. Stumptner et al. show a model-based vs value based approach [4],[35]. A model-based approach is based on providing a representation of the correct behavior of a software system. It consists of behavior rules and responsibility assigned to components. A value-based model has a finer granularity in comparison with the dependency-based model. The value-based model can be used to locate typical errors of fault assignment:

```
p2=p1; // Should be p1.copy() (when p1,p2 variables are initialized)
```

and wrong variable use:

```
y=2*tmp; // Should be y=2*x (when x,y,tmp are initialized)
```

In some situations a bug reveals itself rarely or after a long (two hour) period. To replicate such bug, another approach with tracing is found: by rolling back the program to a previous state and re-execute the buggy region. Srinivasan et al. use this approach in Flashback, which is an OS extension [34]. It uses shadow processes for rollback and replay support. This support is provided by checkpoints and logging. When a checkpoint of a program is created, Flashback creates a shadow process which records the program state in memory. The shadow process suspends until it is used for retrieving the checkpoint. The rollback

option restores the program's checkpoint by overwriting the current program with the program state from the checkpoint. The replay option replays the program from a checkpoint by simulating the OS environment: every system call to files, memory and network is simulated. The simulation is accomplished by logging the system calls and its results from the point when the checkpoint was created. The Flashback primitives: `checkpoint`, `rollback` and `replay` can be used in the source code of the program or by a debugger.

The value-based model provides debugging support on statement level. While this could provide a much faster solution to solve bugs, bugs that are caused by design flaws or are only revealed in rare cases could remain hidden. More research is required to show the benefits of debugging support at statement level. With Flashback a concept of checkpoints is introduced to support debugging. A feature that incorporates a support for accessing history states of debugged programs, could improve debugging performance by removing the overhead of restarting a debugged program.

7.3 Program understanding

This section contains other approaches from the domain of program understanding that contain software visualization.

The Visual Debugger contains multiple levels of detail in its visualization and filter. When changing the detail level to a lower setting, some details are decreased in favor of more overview. Kittilä uses an approach in which the highest level of detail is classes and the lowest level of detail is projects [15]. This approach represents source code dependencies in a dependency structure matrix (DSM). Instead of a dependency graph, a dependency matrix is a matrix in which each entry indicates a dependency. One advantage over a graph is that matrixes are better resistent (no clutter) to a high amount of dependencies. The values of the DSM can be used to provide an indication about structural information such as coupling and cohesion. Optimizations of the DSM are also possible: for example use rows clustering (order of rows) to increase pattern recognition. Another purpose for DSM is to use a lower triangular to indicate distinct cyclic dependencies or use for refactoring (to optimize the dependency structure).

Another program understanding tool with software visualization is AgileJ.[5] This is a commercial Eclipse plug-in. It uses the visualization option of UML class diagrams. These diagrams can be highly customized: filter options for dependencies (hide fields) and also highlight options (distinguish interfaces).

Cornelissen et al. developed Extravis, a visual tool to display execution traces of large programs [7]. The visualization consists of circular bundles within a dependency ring. The bundles represent call relations and are used to reduce visual clutter. The bundles can be

[5] `http://www.agilej.com/reverse-engineered-java-class-diagrams.html`

concentrated to show related dependency calls. This program understanding concept is based on the concept of: related information should be viewed in close proximity. The properties of the bundles are also visualized: the bundle thickness represents the amount of calls and the bundle color indicates the direction of the call, or time in the sequence of the trace. The purpose of indicating the direction of the call is for an overview of the incoming and outgoing calls. A commercial version is also available as part of a reverse engineering environment called SolidFX [37].

Computer programs are far more read than written. Presenting source code with more readability could improve program understanding. Collberg et al. approach is formatting text layout of source code to improve the readability with their tool ART [5]. The idea is to separate control and data flow by representing control flow graphically and data manipulation textually. The advantage lies in combining syntactic rendering with semantic information. The graphical implementation representing control flow includes graphical arrows indicating the direction of the control flow.

For program understanding the tools contain diagrams with high abstraction, are highly customizable, contain concentrated colored bundles and contain a graphical enhanced text layout. This diversity shows the great amount of different directions for possible optimizations to support program understanding and debugging.

Chapter 8

Conclusions

This chapter summarizes the findings of the approach in visual debugging and the resulting evaluation. In addition, the contributions to science are listed and future work is presented.

8.1 Conclusions

It is not uncommon that debugging is performed on unfamiliar software. To support better program understanding for debugging, the approach focuses on visual debugging. The solution is searched from the domains of program understanding, software visualization and debugging. Program understanding is complex because of the amount of source lines and interrelated associations. Using a visual solution such as graphs, supports in accessing large structured data. Abstraction is provided by using a nested graph with allows to hide inner nodes. Distinguishing graph elements visually is accomplished by using different colors and different transparency levels. To reveal the location of bugs, supporting debugging should be focussed on the causality and context of bugs. The approach for supporting debugging is found in using multiple concepts:

- Visualize source code in a graph that provides multiple detail levels
- Synchronize its view with the debugger and optimize the view with filter and highlight options
- Customize the neighborhood and filter settings to support user preferences and multiple debug scenario's

These concepts are implemented in the prototype tool Visual Debugger. This allows to test the concepts for debugging support. The evaluation is conducted by an experiment to rate the value and usefulness of the supporting debugging concepts. With the analyses of the experiment's results, the research questions are answered. Question 1 is answered with input from question 2 and 3. Therefor the first question is question 2.

- **Question 2:** What are key points for supporting debugging?

The key points are retrieved from the list of most used features in debuggers. To support debugging the following features should be supported: close inspection such as stepping and retrieving dependencies such as the feature 'Call hierarchy'. The following question is question 3.

- **Question 3:** Is the use of debugger for program understanding (general) *and* debugging (specific)?

The Visual Debugger purpose is add program understanding to support the debugger. Participants in the experiment expected noticeable program understanding support but were slightly less enthusiastic after the experience in the tasks. The debugging support was rated higher in the experience than the expectation. From the conclusions of question 2 and 3, follows the next question which is question 1.

- **Question 1:** Can local neighborhood visualization be used for supporting debugging?

The visualization of dependencies surrounding the current executed method does provide some added value in program understanding. For the moment this value is not sufficient convincing, but the positive user rating for providing context for breakpoint locations shows that the visualization enhance debug capabilities. Optimization in the supporting debugging concepts could lead to more effective results and a more distinct added value.

8.2 Contributions

This research assignment contributes to the field of visualization for supporting debugging in the following ways:

Thesis: Approach of visual supporting debugger concepts From the supporting debugging concepts to the tool's design to the experiment's design and the evaluation. The concepts include approaches for automating graph navigation, graph filtering, debugger communication and customization.

Tool: The Visual Debugger This consists of the approach in the design and implementation of the tool. The result is a deployed tool integrated as Eclipse plug-in. The plug-in is build as an extension of DA4Java. The implementation consists of the Eclipse preference page, the Eclipse plug-in connection, the supporting debugging concepts, the graph visualization and its optimizations.

Experiment: Preliminary pre-posttest user study This consists of the setup of goals and tasks to inquire the research questions. In addition, the setup of the questionnaires and its approach in design and goals. The tasks include the selection of the target project, target bugs and tasks description. The questionnaires are divided into categories covering the debugger use and experience, tasks and tool evaluation. The results are presented in charts for an easy interpretation. The analyses provide input to answer the research questions.

8.3 Future Work

Within the scope of the thesis not all ideas for supporting debugging could be implemented and evaluated. The ideas are divided into the category of features of the tool and approach of the experiment.

8.3.1 Debug supporting features

The following three features are not implemented in the tool:

Dynamic dependencies support Some Object-Oriented techniques (i.e. polymorphism) in Java allows defining the types of objects when executing the program. Those types cannot be unambiguously be extracted from the source code and its resulting dependencies are not represented in the visualization of the Visual Debugger. Not showing the dynamic dependencies result in an incomplete dependency graph. A solution to this problem is to trace method calls and visualize those dynamic dependencies in the graph.

Call history support Showing the *current* executing method and its neighborhood dependencies adds support in program understanding. But there are situations in which the *previous* method calls are of interest. One solution is to provide highlighting of previous called methods. This provides an easy access to instantaneously recognize the call history. The call history information can be retrieved from the (debugger's) stack frame and from tracing. A challenge is to provide distinct representations between the call methods to indicate the execution order of those methods.

Source code modification support When debugging a program, making a small (or large) modification in the source code is not exceptional. In the current implementation of the Visual Debugger, source code modification has to be manually updated to propagate in the visualization. For improving the usability of the tool this process should be automated. A begin for this feature is already been made but could not be completed before the evaluation in the experiment.

The features 'Dynamic dependencies support' and 'Call history support' were also mentioned for feature requests by the participants of the experiment.

8.3.2 Experiment design

Besides supporting debugging features in the tool, the evaluation in the experiment can also be extended.

Extended debug tasks Debugging and program understanding takes a lot of time. For a better view of the supporting debugging impact, evaluating more tasks divided over a longer period of time is required. This provides the users more time to get adjusted to use a supporting debugging tool for effective debugging.

Comparison of tools With a comparison between other supporting debugging solutions the debug performance can be evaluated. Are other tools more or less efficient? Another question is, how is this performance related to different debug problems (for example: the different complexity of bugs).

Bibliography

[1] E.R. Babbie. *The practice of social research*, chapter Experiment. Wadsworth Belmont, 11th edition, 2007.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 259–268, Washington, DC, USA, 2005. IEEE Computer Society.

[3] G. Canfora, L. Mancini, and M. Tortorella. A workbench for program comprehension during software maintenance. In *Proceedings of the Fourth Workshop on Program Comprehension*, pages 30–39, Mar 29-31 1996.

[4] R. Chen, D. Köb, B. Peischl, and F. Wotawa. Static and dynamic analysis in automated debugging. *MONET Workshop on Model-Based Systems at ECAI*, August 2004.

[5] C.S. Collberg, S. Davey, and T.A. Proebsting. Language-agnostic program rendering for presentation, debugging and visualization. In *Proceedings of the 2000 IEEE International Symposium on Visual Languages*, VL '00, pages 183–190, Washington, DC, USA, 2000. IEEE Computer Society.

[6] T.A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, June 1989.

[7] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J.J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81:2252–2268, December 2008.

[8] A. Dunsmore. Comprehension and visualisation of object-oriented code for inspections. Technical report, University of Strathclyde, Glasgow, UK, 1998.

[9] M. Eisenstadt. Tales of debugging from the front lines. In *Empirical Studies of Programmers, 5th Workshop*, pages 86–112, Norwood, NJ, USA, 1993. Ablex Publishing Corparation.

[10] S. Few. Visualizing change: An innovation in time-series analysis. White paper, SAS, Berkeley, CA, USA, 2007.

[11] J. Goerzen. Finding stubborn bugs with meaningful debug info. *Linux Journal*, 129, 1 2005. http://www.linuxjournal.com/article/7720. Online article accessed 07/28/2011.

[12] R.B. Haber. Visualization techniques for engineering mechanics. *Computing Systems in Engineering*, 1(1):37–50, 1990.

[13] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[14] H. Häuser. *Scientific Visualization: The Visual Extraction of Knowledge from Data*, chapter Generalizing Focus+Context Visualization, pages 305–327. Springer-Verlag Berlin Heidelberg, Secaucus, NJ, USA, 2005.

[15] K. Kittilä. Analysing and managing software dependencies with a dependency structure matrix tool. Master's thesis, University of Oulu, 2008.

[16] A.J. Ko. Debugging by asking questions about program output. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 989–992, New York, NY, USA, 2006. ACM.

[17] A.J. Ko and B.A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[18] M. Kroening. Eclipse: A developer's story. *Dr.Dobb's*, 2003. http://drdobbs.com/184401695?pgno=1. Online article accessed 11/11/2010.

[19] J.H. Larkin and H.A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive science*, 11:65–99, 1987.

[20] M.M. Lehman. Laws of software evolution revisited. In *Proceedings of the 5th European Workshop on Software Process Technology*, volume 1149 of *EWSPT '96*, pages 108–124, London, UK, 1996. Springer-Verlag.

[21] H. Lieberman and C. Fry. Bridging the gulf between code and behavior in programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '95, pages 480–486, New York, NY, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[22] B.P. Lientz, E.B. Swanson, and G.E. Tompkins. Characteristics of application softeware maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[23] R. Likert. A technique for the measurement of attitudes. *Archives of Psychology*, 140:1–55, 1932.

[24] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, April 1976.

[25] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *8th International Workshop on Principles of Software Evolution*, pages 13–22, Washington, DC, USA, Sept. 5-6 2005. IEEE Computer Society.

[26] G.A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 63(8):1–97, 1956.

[27] T.G. Moher. Provide: a process visualization and debugging environment. *Software Engineering, IEEE Transactions on*, 14(6):849–857, June 1988.

[28] P. Mutzel and P. Eades. *Software Visualization*, chapter 4 Graphs in Software Visualization, pages 285–294. Springer Berlin / Heidelberg New York, 2002.

[29] S. Nellen. Eclipse plugin for tinyos debugging. Technical report, Institue of Technology, Zürich, CH, 2009. Semester Thesis.

[30] W. De Pauw, N. Mitchell, M. Robillard, G. Sevitsky, and H. Srinivasan. Drive-by analysis of running programs. In *Proceedings of the ICSE Workshop on Software Visualization*, pages 17–22, 2001.

[31] M. Pinzger, K. Gräfenhain, P. Knab, and H.C. Gall. A tool for visual understanding of source code dependencies. *The 16th IEEE International Conference on Program Comprehension*, 0:254–259, 2008.

[32] V. Rajlich. Intensions are a key to program comprehension. In *International Conference on Program Comprehension (ICPC '09)*, pages 1–9, Washington, DC, USA, May 17-19 2009. IEEE Computer Society.

[33] S. Ratanotayanon and S.E. Sim. Inventive tool use to comprehend big code. *IEEE Software*, 25(5):91–92, 2008.

[34] S.M. Srinivasan, S. Kandula, C.R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *In USENIX Annual Technical Conference, General Track*, pages 29–44, 2004.

[35] M. Stumptner, D. Wieland, and F. Wotawa. Comparing two models for software debugging. In *Proceedings of the Joint German/Austrian Conference on AI: Advances in Artificial Intelligence*, KI '01, pages 351–365, London, UK, 2001. Springer-Verlag.

[36] T. Systä, K. Koskimies, and H.A. Müller. Shimba - an environment for reverse engineering java software systems. *Software: Practice and Experience*, 31(4):371–394, 2001.

[37] A. Telea and L. Voinea. Solidfx: An integrated reverse engineering environment for c++. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 320–322, april 2008.

[38] L. Vogel. Eclipse plugin development tutorial. *Java, Eclipse, Android, and Web Programming Tutorials*, 2008. http://www.vogella.de/articles/EclipsePlugIn/article.html. Online article accessed 11/12/2010.

[39] A. von Mayrhauser and A.M. Vans. Comprehension processes during large-scale maintenance. *Proceedings of the 16th International Conference on Software Engineering*, pages 39–48, 1994.

[40] A. von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[41] M. de Wit. Managing clones using dynamic change tracking and resolution. Master's thesis, TU Delft, 2008.

[42] A. Zeller. Visual debugging with ddd. *http://www.drdobbs.com/tools/184404519*, 3 2001. Online article accessed 8/12/2010.

[43] A. Zeller and D. Lütkehaus. Ddd—a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.

Appendix A

Documentation of the experiment

This chapter contains the hand-outs used during the experiment. First, the pretest questionnaire. For executing the task, the task and tutorial documentation was presented. After finishing the tasks, the posttest questionnaire was presented.

A.1 Pretest questionnaire

Visual Debugging

Introduction

A part of developing software is verifying the execution of the source code according to its expectation. When the program behaves differently than expected, the developer has to find the cause somewhere in the source code. The main tool to inspect the source code and follow the program's execution is the debugger. Because you have to guess the starting point of debugging and path to follow, using the debugger to solve debug problems is a time consuming process. A way to support the debugger by narrowing down the amount of source code to inspect should greatly reduce the duration of the process. An approach is to present a visualization of the source code to support choosing the debug strategy.

The idea of visualization of the source code is the basis of the tool Visual Debugger. The tool shows the static dependency structure of the source code in a graph. To support the debugging process, only the neighborhood of the execution point is shown in higher detail than its surroundings.

The experiment

In this experiment the Visual Debugger is tested inside the Eclipse environment. The tasks involve solving debug problems with the support of Visual Debugger. The experiment starts with a couple of questions to get an impression of the development experience and expectation for the tool. After the questions a few tasks are presented to test the tool. After the tasks another series of questions will be asked about the tool experience.

1

Figure A.1: Pretest questionnaire (page 1 of 4)

Pretest questionnaire

In this questionnaire you will be ask about your software development experience, debugging experience and expectations for the Visual Debugger tool. Please fill in the following statements according to your opinion. For the validity of the experiment it is important that you answer true fully and not just positively towards the tested tool.

A. Software Development Experience

Below a number of general statements about software development are shown. Please rate each of the statements on a scale from 1 (strongly disagree) to 5 (strongly agree) to indicate to what extent they apply to you.

	1	2	3	4	5
I consider myself an experienced Java developer	O	O	O	O	O
I often use Eclipse for writing java code	O	O	O	O	O
I am familiar with JHotdraw	O	O	O	O	O

B. Attitude toward Debugging

The Visual Debugger tool is build as an extension to the Eclipse debugger. We like to know more about how you use the debugger.

The following statements are used to indicate your personal experience with debugging. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
I often use the debugger to find bugs faster	O	O	O	O	O
I often use the debugger to understand a piece of source code	O	O	O	O	O
I often use the Eclipse debugger	O	O	O	O	O
I often use alternative methods for debugging, such as println(...) or logging	O	O	O	O	O

Do you often use additional tools for debugging?

- If yes, which tools are you using?

2

Figure A.2: Pretest questionnaire (page 2 of 4)

C. Debugging Approach

Debug problems can be approached in many ways. For more insight in your approach for debug problems, we like to know how you use typical debugger functionality.

1. Finding starting points for debugging

The following statements are used to indicate how you perform debugging. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
I often use test cases as starting points for debugging	O	O	O	O	O
I often use action handlers as starting points for debugging	O	O	O	O	O
I often use the main method as starting point for debugging	O	O	O	O	O
I often use stack traces as starting point for debugging	O	O	O	O	O

Do you often use other information as starting points for debugging?

- If yes, which ones are you using?

2. Using the debugger

The following statements are used to indicate how you perform debugging. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
I often use breakpoints to stop the execution and inspect the state of objects	O	O	O	O	O
In Eclipse, I often use the "Watch Expression" feature to inspect the state of objects	O	O	O	O	O
I often use the stepping-functionality of debuggers to step through the execution	O	O	O	O	O
I often use the (Eclipse) Type Hierarchy feature to inspect the type hierarchy of variables	O	O	O	O	O
I often use the (Eclipse) Call Hierarchy feature to inspect the incoming and out-going calls of methods	O	O	O	O	O

3

Figure A.3: Pretest questionnaire (page 3 of 4)

D. Debugging expectations

1. Expectations for a Tool like Visual Debugger

The final question of this survey is about your expectations for a tool that helps to debug source code. Please read the description below and then rate each of the following statements on a scale from 1 (strongly disagree) to 5 (strongly agree):

"The Visual Debugger is an extension to the Eclipse debugger that visualizes the dependency graph of the execution point while debugging using a graph-based visualization. Nodes in the graph present packages, classes, interfaces, methods, and attributes. Edges in the graph present the dependencies between these entities, such as method calls, attribute accesses, and class inheritance. An execution point is defined as the method which is currently debugged. The tool mainly visualizes the incoming and outgoing dependencies of the execution point and thereby allows the developer to inspect the dependencies and the dependent entities. The dependency graph and visualization is updated whenever the debugger is suspended, which occurs at breakpoints or through stepping."

2. Expectations for program understanding

	1	2	3	4	5
Such a tool will significantly help in understanding the program	O	O	O	O	O
Showing the neighborhood of the execution point will help in understanding the program	O	O	O	O	O
The additional user interactions for the tool will likely hinder program understanding	O	O	O	O	O
I would use such a tool extensively for understanding programs	O	O	O	O	O

The main purpose of the Visual Debugger is to support debugging through visualization. Please rate the following statements from strongly disagree (1) to strongly agree (5):

3. Expectations for support debugging

	1	2	3	4	5
Such a tool will significantly help to ease debugging	O	O	O	O	O
Showing the neighborhood of the execution point will help in debugging	O	O	O	O	O
The additional user interactions for the tool will likely hinder debugging	O	O	O	O	O
I would use such a tool extensively for debugging	O	O	O	O	O

4

Figure A.4: Pretest questionnaire (page 4 of 4)

A.2 Tasks

Tasks

Introduction

The tasks in this experiment involve the open source project JHotdraw. This framework will be used to tests the ability of Visual Debugger to support program understanding in the debug process.

JHotdraw is an open source modeling framework. The software is build as an educational tool. One of its purposes is a showcase of design patterns such as MVC, Observer, Adapter, Composite, Decorator, and Strategy.

The execution of the tasks requires the availability of the source code. The Eclipse environment which is setup for this experiment contains the source code of JHotdraw version 6b1. Besides the source code of the framework, several demonstration applications exist. These applications show how the JHotdraw framework should be implemented.

For the task we use the demonstration application: JavaDraw. JavaDraw is an object-based drawing program: it draws figure objects and all user commands are applied on those figure objects.

The starting point for JavaDraw in the source code is: org.jhotdraw.samples.javadraw.JavaDrawApp, and is already set as a launch configuration.

Task 1: "Align"

This task involves finding the cause of a bug. To begin this task start with running JavaDraw . The maximum time limit for this task is 20 minutes.

Bug description:

Before opening a new drawing using one of the 'align' commands from the menu will cause an exception.

The objective is to find the problem causing this bug. To help you start the debug process, a breakpoint (in org.jhotdraw.util.CommandMenu) is already set at the point the 'align' command is executed.

Hint:

To solve this problem you should use the breakpoint as a starting point and start the Visual Debugger. Use additional breakpoints to inspect other interesting parts of the source code.

Figure A.5: Task (page 1 of 2)

Answer:

When you found the cause of the bug, please write a short description of the cause and a possible solution of the bug. Also export the breakpoints to a file (e.g. “task1.bkpt”). The export breakpoint functionality is located in the breakpoint list and accessed through the context menu:

Task 2: "Undo"

This task also involves finding the cause of a bug. To begin this task start with running JavaDraw . Create two figures in a new drawing, select them (with the mouse or shift-key) and use the 'group' command to group the figures. The result is two figures transformed as one group figure. The maximum time limit for this task is 40 minutes.

Bug description:

A fill color operation on a group figure cannot be undone; the undo operation does not work as intended.

The objective in the tasks is to find the problem causing this bug. You can use the same starting point (the breakpoint in CommandMenu) as the previous task. Because this breakpoint will trigger on all commands in the menu, you should enable it after you executed the fill color command.

Answer:

When you found the cause of the bug, please write a short description of the cause and a possible solution of the bug. If you have not found the cause of the bug, please describe how the 'undo' command for this task works. Also export the breakpoints used in this task to a file:

Figure A.6: Task (page 2 of 2)

A.3 Posttest questionnaire

Posttest questionnaire

This questionnaire will question you about your experience with the Visual Debugger. Please fill in the following statements according to your opinion. Keep in mind to answer true fully.

A. Tasks assessment

The first part of statements considers the tasks. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
The tasks presented realistic debugging tasks	O	O	O	O	O
The tasks were too difficult for me	O	O	O	O	O
The tasks were very interesting to do	O	O	O	O	O
I feel enthusiastic about the tasks	O	O	O	O	O
I felt a lot of time pressure	O	O	O	O	O
I would have needed more guidance in completing the tasks	O	O	O	O	O

B. Tool user experience

In executing the tasks you were encouraged to use the Visual Debugger tool. We like to know your experience with the tool, especially the usability and functionality.

1. Usability for program understanding

The next part of statements considers the Visual Debugger for program understanding. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
Visual Debugger will significantly help in understanding the program	O	O	O	O	O
Showing the neighborhood of the execution point will help in understanding the program	O	O	O	O	O
The additional user interactions for Visual Debugger will likely hinder program understanding	O	O	O	O	O
I would use Visual Debugger extensively for understanding programs	O	O	O	O	O

1

Figure A.7: Posttest questionnaire (page 1 of 3)

2. Usability for supporting debugging

The next part of statements considers the Visual Debugger for supporting debugging. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
Visual Debugger will significantly help to ease debugging	O	O	O	O	O
Showing the neighborhood of the execution point will help in debugging	O	O	O	O	O
The additional user interactions for Visual Debugger will likely hinder debugging	O	O	O	O	O
I would use Visual Debugger extensively for debugging	O	O	O	O	O

3. Tool features

The next part of statements considers the Visual Debugger settings. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
The setting for change in dependency level is useful	O	O	O	O	O
- Which level did you used the most?					
The setting for change in transparency level is useful	O	O	O	O	O
The stepping support is useful	O	O	O	O	O
The filter setting is useful	O	O	O	O	O
- Which level did you used the most?					

4. GUI handling

The next part of statements considers the Visual Debugger graph handling. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
The graph of the tool required much user navigation such as panning and zooming	O	O	O	O	O
The filtering and adding (dependencies) of nodes worked as expected	O	O	O	O	O
The filtering and adding (dependencies) of nodes was often used	O	O	O	O	O
The amount of screen space of the tool was sufficient	O	O	O	O	O

2

Figure A.8: Posttest questionnaire (page 2 of 3)

C. Conclusion

The next part of statements considers the final verdict about the Visual Debugger. Please rate the following statements from strongly disagree (1) to strongly agree (5):

	1	2	3	4	5
Graphical presentation of source code such as Visual Debugger has added value for debugging	O	O	O	O	O
Manipulation of the graph is intuitive	O	O	O	O	O
Visual Debugger is easy to use	O	O	O	O	O
Visual Debugger is ready for everyday use	O	O	O	O	O
Visual Debugger is a useful extension to the debugger	O	O	O	O	O

Comment

Thank you for participating in this experiment. Please write down any last comments you have about the experiment you wish to share:

3

Figure A.9: Posttest questionnaire (page 3 of 3)

A.4 Tutorial

Tutorial for the Visual Debugger

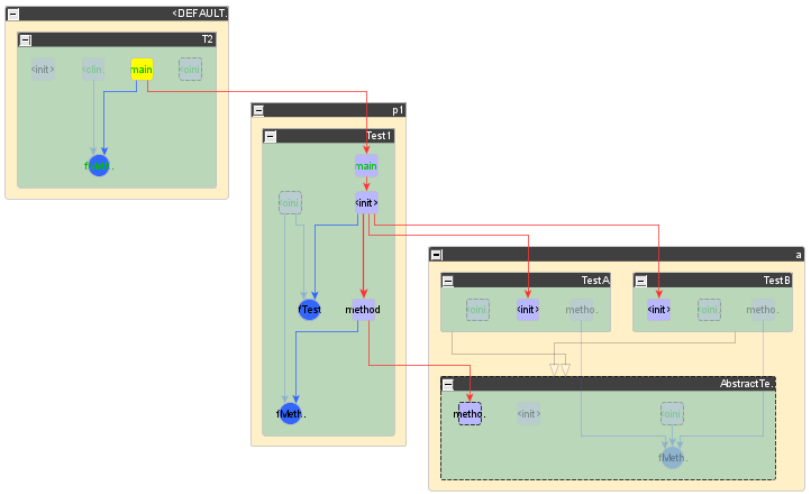
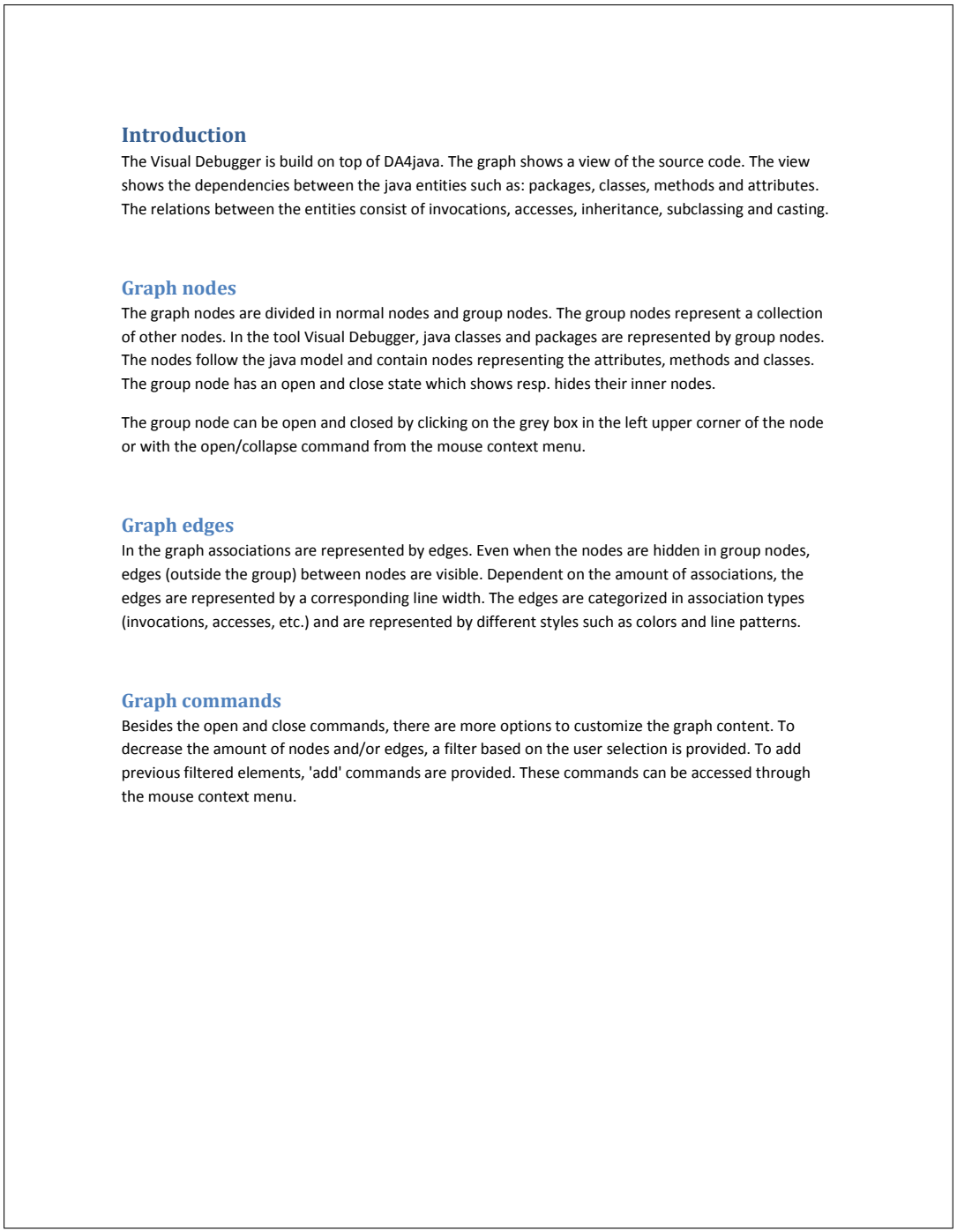


Figure 1. Dependency structure for the execution point of the method 'main'.

Figure A.10: Tutorial (page 1 of 4)



Introduction

The Visual Debugger is build on top of DA4java. The graph shows a view of the source code. The view shows the dependencies between the java entities such as: packages, classes, methods and attributes. The relations between the entities consist of invocations, accesses, inheritance, subclassing and casting.

Graph nodes

The graph nodes are divided in normal nodes and group nodes. The group nodes represent a collection of other nodes. In the tool Visual Debugger, java classes and packages are represented by group nodes. The nodes follow the java model and contain nodes representing the attributes, methods and classes. The group node has an open and close state which shows resp. hides their inner nodes.

The group node can be open and closed by clicking on the grey box in the left upper corner of the node or with the open/collapse command from the mouse context menu.

Graph edges

In the graph associations are represented by edges. Even when the nodes are hidden in group nodes, edges (outside the group) between nodes are visible. Dependent on the amount of associations, the edges are represented by a corresponding line width. The edges are categorized in association types (invocations, accesses, etc.) and are represented by different styles such as colors and line patterns.

Graph commands

Besides the open and close commands, there are more options to customize the graph content. To decrease the amount of nodes and/or edges, a filter based on the user selection is provided. To add previous filtered elements, 'add' commands are provided. These commands can be accessed through the mouse context menu.

Figure A.11: Tutorial (page 2 of 4)

Visual Debugger settings

The following settings are designed to customize the neighborhood feature of Visual Debugger. The settings are accessible through the property page of the project.

Dependency level

The level of dependency indicates which dependency depth level from the current execution point is shown: in the graph the classes and packages representations, which contain the dependent entities, are ‘opened’ and the whole dependency structure is visualized more clear (less transparent). A setting of ‘level 0’ shows no incoming and outgoing dependencies, a setting of ‘level 1’ shows 1 depth level of incoming and outgoing dependencies (entity – association – entity).

Transparency level

To distinguish the difference between the set entities and associations of the dependency structure and the set outside the dependency structure, a transparency setting is introduced. This setting provides the ability to increase or decrease the visual difference between the sets.

Filter level

Large programs contain a great amount of entities and associations. To improve readability and performance, the filter setting provides the ability to reduce the amount of graph elements. The dependency level setting determines the size of the dependency structure. Based on this dependency structure, the filter filters entities and associations outside the dependency structure. The 'package' setting will remove packages which contain no dependency elements. The 'Class' and 'Method' settings filter incrementally more elements.

Figure A.12: Tutorial (page 3 of 4)

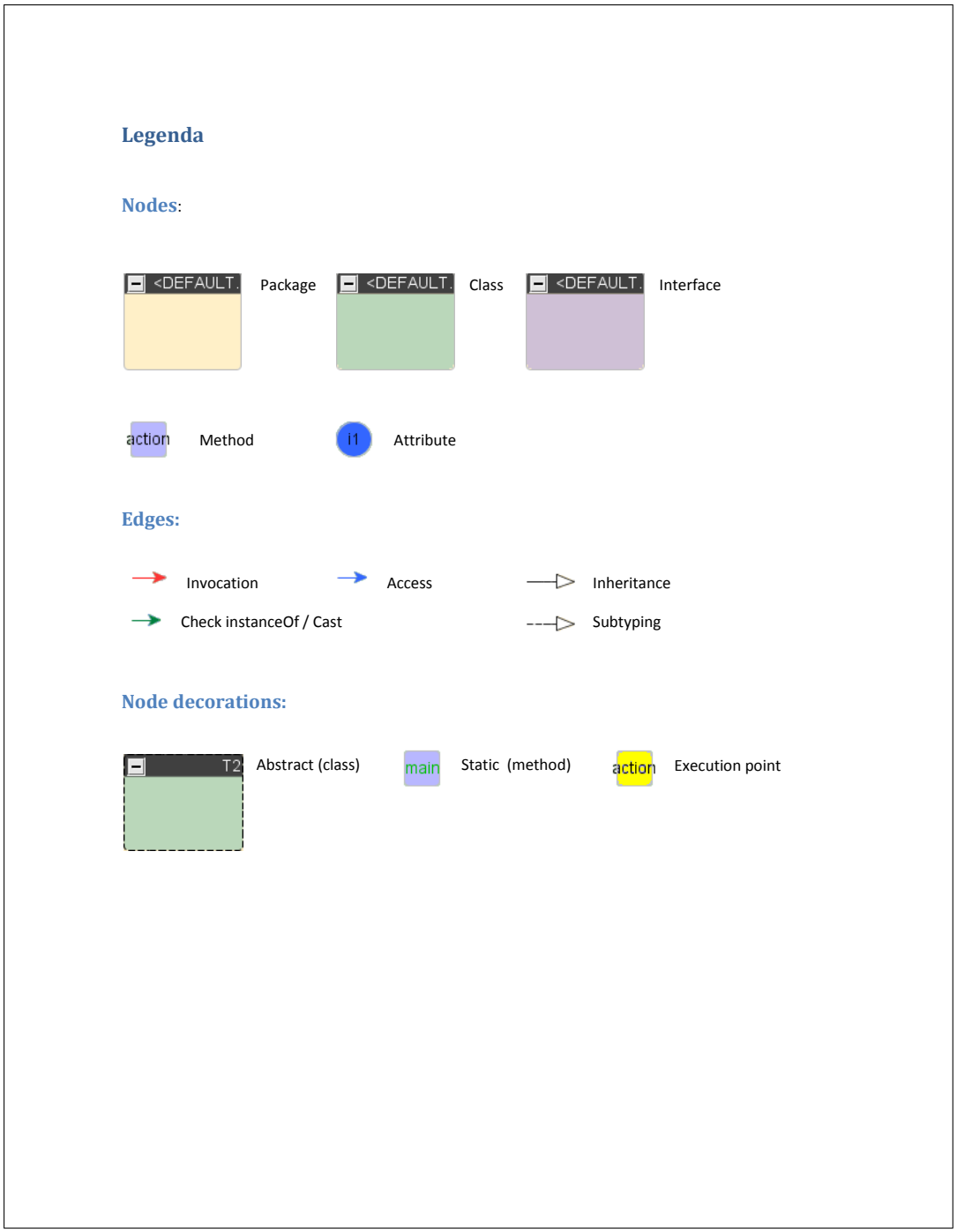


Figure A.13: Tutorial (page 4 of 4)