DANILO MUTTI

Coverage based debugging visualization

São Paulo

2014

DANILO MUTTI

Coverage based debugging visualization

Dissertation presented to the School of Arts, Sciences and Humanities, University of São Paulo to achieve the title of Master of Science by the Graduate Program in Information Systems.

Corrected version containing the amendments requested by the judging commission on October 31, 2014. The original version is available at the reserved collection in the Library of EACH-USP and in the Digital Library of Theses and Dissertations of USP (BDTD), in accordance with resolution CoPGr 6018, October 13, 2011.

Supervisor: Prof. Dr. Marcos Lordello Chaim

São Paulo

2014

CATALOGING IN PUBLICATION (CIP) DATA
(University of Sao Paulo. School of Arts, Sciences and Humanities. Library)

Mutti, Danilo
Coverage based debugging visualization / Danilo Mutti ; supervisor, Marcos Lordello Chaim. – Sao Paulo, 2014
137 p. : ill.

Dissertation (Master of Science) - Graduate Program in Information Systems, School of Arts, Sciences and Humanities, University of Sao Paulo, Sao Paulo, 2014
Corrected version

1. Software testing and evaluation. 2. Software verification and validation. 3. Software development. 4. Error (Computational failures). I. Chaim, Marcos Lordello, sup. II. Title.

CDD 22.ed. – 005.14

Approval Sheet

Master's Dissertation entitled *"Coverage based debugging visualization"*, defended by Danilo Mutti and approved on October 31, 2014 in São Paulo, State of São Paulo, by the judging commission composed of:

Prof. Dr. Marcos Lordello Chaim
Chairman
School of Arts, Sciences and Humanities
University of São Paulo

Prof. Dr. Manoel Gomes de Mendonça Neto
Department of Computer Sciences
Federal University of Bahia

Prof. Dr. João Luiz Bernardes Jr.
School of Arts, Sciences and Humanities
University of São Paulo

To everyone that supported me through this journey.

Abstract

MUTTI, Danilo. **Coverage based debugging visualization**. 2014. 137 p. Dissertation (Master of Science) – School of Arts, Sciences and Humanities, University of São Paulo, São Paulo, 2014.

Fault localization is a costly task in the debugging process. Usually, developers analyze failing test cases to search for faults in the program's code. Visualization techniques have been proposed to help developers grasp the source code and focus their attention onto locations more likely to contain bugs. In general, these techniques utilize two-dimensional visualization approaches. We introduce a three-dimentional visual metaphor, called *CodeForest*, which represents a software as a cacti forest. In the CodeForest, nodes (sets of statements executed in sequence) are thorns, methods are branches, and classes are cacti. Heuristics—based on the frequency that lines of codes are executed in successful and failing test cases—are used to assign suspiciousness values to the elements (thorns, branches, and cacti) of the forest. The new metaphor was implemented as a plug-in targeted to the Eclipse Platform. This plug-in includes the mapping of suspiciousness values to elements of a virtual forest, a parameterized trimmer, which filters elements based on their score or text, and a list of most suspicious methods (also known as "roadmap"), to guide the developer on his/her debugging session. An exploratory experiment was conducted; the results indicates that the tool supports developers with and without experience. Users with low or no experience utilized the roadmap and the virtual 3D environment to investigate the defect. More experienced users prefer to use the roadmap as a guide to narrow which parts of the source code should be explored.

Keywords: Software visualization. Visualization of debugging information. Fault localization. Code coverage.

Resumo

MUTTI, Danilo. **Visualização de depuração baseada em cobertura**. 2014. 137 f. Dissertação (Mestrado em Ciências) – Escola de Artes, Ciências e Humanidades, Universidade de São Paulo, São Paulo, 2014.

Localizar falhas é uma tarefa custosa do processo de depuração. Normalmente, os desenvolvedores analisam casos de teste que falham para procurar por defeitos no código fonte de um programa. Técnicas de visualização têm sido propostas para ajudar os desenvolvedores a entender o código fonte e focar sua atenção nos locais com a maior probabilidade de conterem defeitos. Geralmente, essas técnicas utilizam abordagens de visualização bidimensional. Nesse trabalho é introduzida uma metáfora visual em três dimensões, chamada *CodeForest*, que representa um programa como uma floresta de cactus. Na *CodeForest*, nós (conjunto de comandos executados em sequência) são representados como espinhos, métodos como galhos e classes como troncos. Para associar valores de suspeição aos elementos da floresta (espinhos, galhos e troncos) utilizam-se heurísticas, baseadas na frequência com que linhas de código são executadas em casos de teste finalizados com sucesso e com falha. A nova metáfora foi implementada como um complemento da plataforma Eclipse de desenvolvimento de programas. Esse complemento inclui o mapeamento dos valores de suspeição para elementos de uma floresta, uma ferramenta de poda parametrizada—que filtra elementos com base em seu texto e valor de suspeição—e uma lista dos métodos mais suspeitos (conhecida como "roteiro") para guiar o desenvolvedor em sua sessão de depuração. Um experimento exploratório foi conduzido e os resultados indicam que a ferramenta apoia a tarefa de depuração tanto de desenvolvedores experientes quanto inexperientes. Usuários com pouca ou nenhuma experiência utilizaram o roteiro e o ambiente virtual 3D para investigar o defeito. Usuários mais experientes preferiram utilizar o roteiro como um guia para restringir quais partes do código fonte deveriam ser exploradas.

Palavras-chave: Visualização de software. Visualização de informação de depuração. Localização de defeitos. Cobertura de código.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

During the software development activity, there is a major concern in building programs that can be executed correctly without any failures. Such failures are the visible consequence of defects hiding in the source code. Introduced by developers[1] due to human mistakes, some defects are easily corrected; for example: a simple typo; while others are more difficult to fix, such as comprehension issues regarding some feature of the system.

The practice of Software Engineering offers a myriad of testing techniques attempting to reveal possible existing failures. Once such failures have been discovered, the following task is to localize and correct the defect that causes the observable failure. Such a task is called *debugging*.

1.1 Context

Debugging is the activity whose goal is to detect, locate and correct defects present in programs. It is an expensive activity which requires developers to analyze the program's source code as well as its state (stack trace and memory dumps). In practice, developers use print statements in code or breakpoints from symbolic debuggers to inspect the state of program variables in the search for faults. This manual process can be expensive and ad-hoc (JONES; BOWRING; HARROLD, 2007). Recent studies estimate that developers spend half of their programming time debugging, which approximates to a global cost of 312 billion of dollars a year (BRITTON L. JENG; KATZENELLENBOGEN, 2013). Several techniques to automate fault localization have been proposed to improve the debugging process, aiming at reducing effort and time spent (JONES; HARROLD; STASKO, 2002; AGRAWAL et al., 1995; WOTAWA; STUMPTNER; MAYER, 2002; ZELLER, 2002; RENIERIS; REISS, 2003).

In particular, some techniques utilize coverage information of code components such as statements, predicates, def-use associations and call functions to automate the

[1] In this research, the term developer encompasses the roles that are directly involved in the lifecycle of a failure: analysts, programmers, and maintainers.

fault localization process (JONES; HARROLD; STASKO, 2002; AGRAWAL et al., 1995; WOTAWA; STUMPTNER; MAYER, 2002). Heuristics—based on the frequency with which code components are executed in failing and successful test cases—are used to assign suspiciousness values to statements. A list of statements sorted in descending order of suspiciousness is then prepared for the developer. We call these approaches to fault localization *coverage based debugging*.

Coverage based debugging information, however, should be presented to the developer to be useful. An obvious strategy is to list the program's statements ordered by their values of suspiciousness. Another strategy uses graphical resources to highlight fault-prone locations. *SeeSoft* (EICK; STEFFEN; JR, 1992) is a technique to visualize large portions of code in which each line of code is displayed as a thin line. In the coverage based debugging context, the suspiciousness value of each line of code defines its hue and brightness (JONES; HARROLD; STASKO, 2002). Figure 1 presents SeeSoft being utilized to visualize a faulty program where the more suspicious locations are colored in bright red. This visualization technique has the advantage of keeping the code structure (e.g., indentation, line length and blank lines), which facilitates the understanding of the code.

Nonetheless, current visualization techniques for coverage based debugging lack contextual information. Suppose that a developer should look for faults in a large software written in Java as the one described in Figure 1. If she had to choose a package, class and method to initiate her investigation which one should she start with? If the defect was not found in the first package, class and method analized, which are the next ones to be investigated? Such an information is not available in 2D visual metaphors like SeeSoft because they do not associate suspiciousness with packages, classes and methods—only with lines of code. In this sense, they fall short of providing a roadmap to guide the developer in the fault localization process.

1.2 Motivation

The motivation of this research resides in the observation that current mechanisms to visualize and inspect debugging information do not provide enough contextual information to guide the developer towards the fault site. Lists of suspicious statements and bidimensional visualization metaphors does not seem to provide the contextual information required for an effective fault localization process. Therefore, new visualization

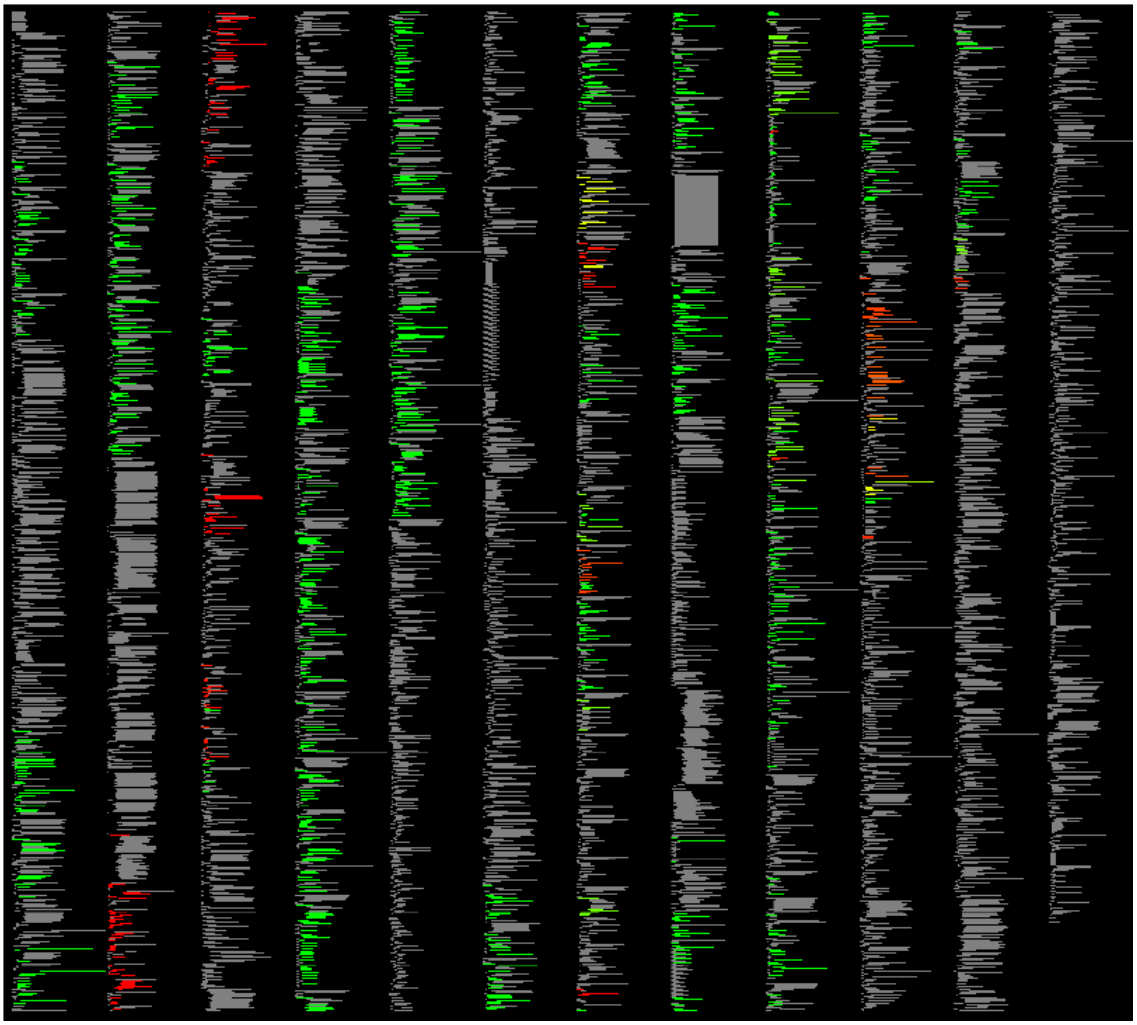


Figure 1 – *SeeSoft utilized to visualize suspicious excerpts of code (obtained from (JONES; HARROLD; STASKO, 2002)).*

metaphors and mechanisms that add contextual information are needed to leverage developer's performance during debugging.

1.3 Objectives

The goal of this work is to develop novel ways to visualize and inspect debugging information. To achieve such a goal, a new visual metaphor to represent debugging information (e.g., the suspiciousness of code excerpts) is proposed.

In addition, mechanisms to inspect and select debugging information to help the developer focus her attention onto pieces of code more likely to contain the fault's site are introduced.

Thus, the overall aim of this research is to improve the developer's experience

in navigating through the information generated by coverage based debugging. A recent study (PARNIN; ORSO, 2011) with developers using coverage based debugging suggests that additional contextual information might leverage developers' performance.

This research specific objectives are defined as follows:

- Develop a novel three-dimensional metaphor to represent suspiciousness information obtained from coverage based debugging techniques.
- Explore how to display larger program entities, namely classes, methods, and their suspiciousness values, in the metaphor.
- Embed the new metaphor as a plug-in into a well established Integrated Development Environment.
- Add contextual information obtained from integration coverage to support fault localization (SOUZA; CHAIM, 2013).
- Explore new avenues to improve user experience.
- Plan and execute an experiment to validate the new metaphor and its implementation as a plug-in.

1.4 Contributions

This research gave origin to several contributions. They are briefly described in what follows.

- A novel three-dimensional metaphor called CodeForest (Chapter 4), which aims at representing suspiciousness data obtained from coverage based debugging techniques (Section 2.2).
- A thorough literature review of 2D and 3D methods to visualize software data (Chapter 3).
- A functional standalone implementation of the CodeForest metaphor (Chapter 4), which worked as a sandbox environment to test and improve the metaphor.
- The CodeForest plug-in (Chapter 5), embedded into a well known Java IDE.
- An exploratory experiment (Chapter 6) to check the adherence of the CodeForest metaphor/plug-in in the activity of investigating defects.

1.5 Key findings

In this research, we conducted an exploratory experiment in which the new metaphor and the features introduced in the plug-in that realizes it were assessed.

We found out that mapping classes into the elements of the new metaphor had a good reception amongst users. Nevertheless, the positioning of the elements representing methods according to their score received divergent evaluations and requires further experiments.

Developers with lower levels of skill in Java are more sympathetic to investigate the codebase utilizing the new metaphor virtual environment. They do utilize contextual information (also known as "roadmap") to filter the elements of the metaphor, but the primary investigation happens in the virtual environment.

More experienced users prefer to solely use the roadmap as an investigation guide, and utilize the filtering tools to check the remaining elements. This way, they are able to assess which elements of the codebase should be inspected and move on directly to the source code.

1.6 Organization

This chapter presented the context, motivation and objectives of our research, whose main objective is to develop novel ways to select, visualize, and inspect debugging information.

The remainder of this document is organized as follows.

- Chapter 2 presents the necessary background in software testing, coverage based debugging, and software visualization.
- Chapter 3 examines the related work.
- Chapter 4 presents a new metaphor for visualizing coverage based debug information, its design rationale, and a prototype software, developed to test the feasibility of our ideas.
- Chapter 5 contains the details required to turn a prototype into a complete tool, embedded in the Eclipse Platform.

- Chapter 6 describes an experiment in which the use of the Eclipse plug-in is assessed.
- Chapter 7 draws conclusions and lists possible extensions to this research.

Chapter 2

Background

This chapter contains the fundamental concepts applied on this research. Section 2.1 presents the concepts related with Software Testing. Section 2.2 describes the concept regarding code coverage, which is the set of of components covered during the execution of test cases. Debugging techniques based on code coverage, the basis of this research, are detailed in Section 2.3. Finally, Section 2.4 covers the necessary concepts related with the fields of Information Visualization and Human-Computer Interaction, both necessary to conceive our novel metaphor, CodeForest.

2.1 Testing

This section presents the concepts defined by the Software Testing discipline applied to automatic debugging strategies. Information generated during the execution of tests enables the identification of failures and, thus, the search for defects.

2.1.1 Bugs everywhere

No one knows when the first logical bug was introduced in a program; however, we do know the exact day when the first actual bug was found in a computer. It was a moth, found in the Harvard Mark II machine and eventually appended to a log book by a technician (Figure 2). On September 9^{th} 1947 the bug found its way into one of the 13,000 high-performance relays commissioned for this particular machine. Instantly killed by the current, the remains of the insect made the machine fail. This incident proved that computer problems could be caused by actual bugs.

The term gained popularity since that episode, causing software engineering practitioners to use the term "bug" to denote several things: an awry line of code, a problematic state detected in runtime (for example, a wrong reference) or an incorrect program execution (a program crashes in the presence of certain inputs). Despite being interconnected—one is a consequence of the other—lines of code, program state, and execution output are elements separated in space and time. Additionally, using a single term

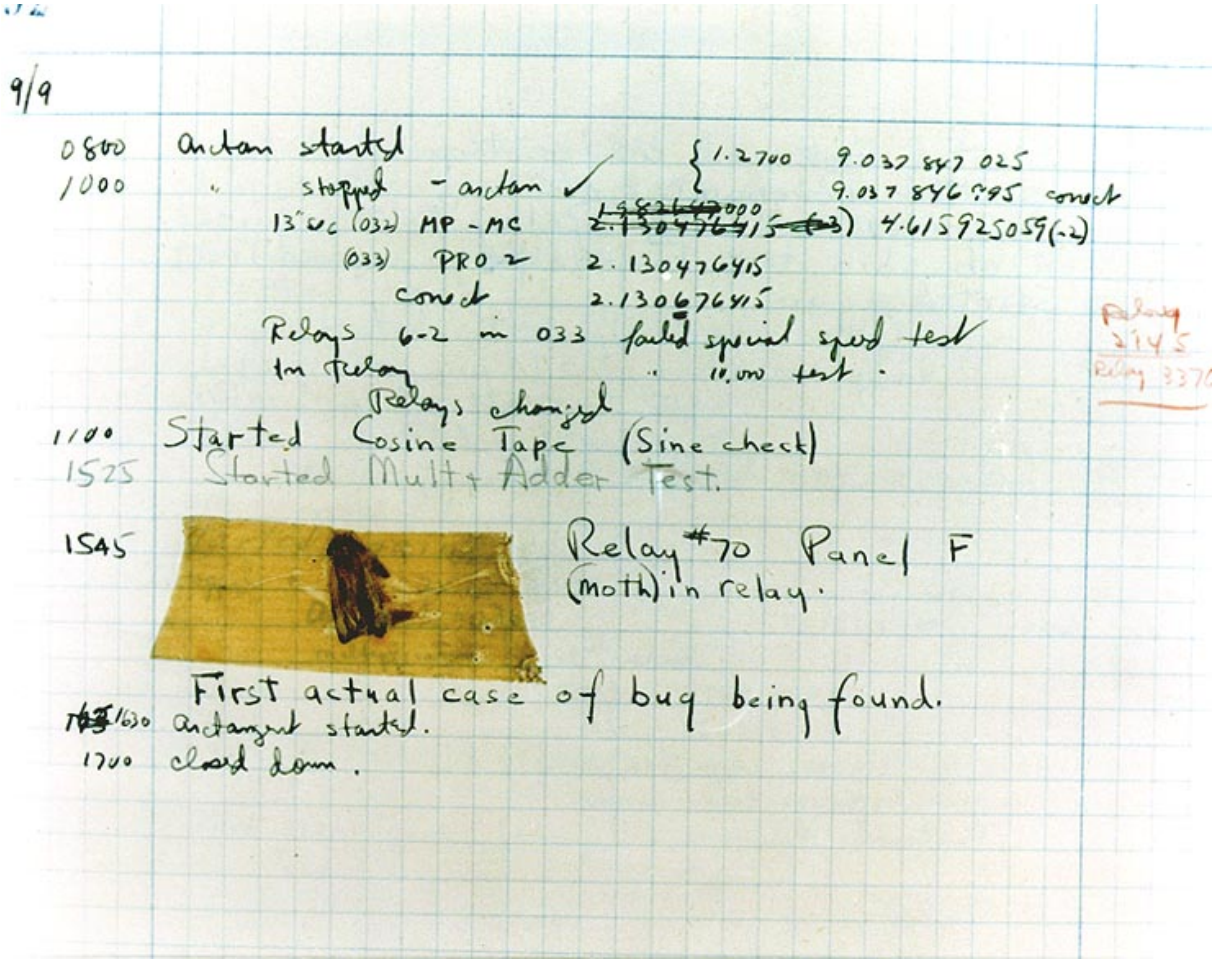


Figure 2 – *The first "computer bug"(CENTER, 1947)*

that assumes a different meaning depending on the context clutters communication and favors confusion and misunderstandings.

For this reason, it is important to disambiguate such situations. Zeller (ZELLER, 2009) proposes these terms:

- *A defect* denotes an incorrect piece of program code.
- *An infection* denotes an incorrect program state.
- *A failure* denotes an observable incorrect program behavior.

The IEEE Software and Systems Engineering Vocabulary[1] (*SEVOCAB*) explains "fault" the same way as "defect"(ZELLER, 2009). It also uses the term "bug" as a synonym for "fault", which makes these terms equivalent.

[1] `http://pascal.computer.org/sev_display/index.action`

The world where actual bugs could cause a failure lies in the past, along with the original definition of debugging. Since then, the term has been used to denominate the activity that locates and corrects faults (defects) in a computer program using one or more techniques, which can include the use of breakpoints, desk checking, dumps, inspection, reversible execution, single-step operation, and traces. It is curious to notice that this definition has not changed in twenty years (IEEE, 1990; IEEE, 2010).

Humphrey (HUMPHREY, 2013) claims a psychological appeal in using the word defect in detriment of bug. According to him, calling defects "bugs" usually creates a tendency to minimize a potentially serious problem. For example, a piece of software that gets to the end of Quality Assurance with "a few bugs". When put this way, it does not sound so harmful. As scarier as it sounds, defects are more like land mines than insects. Not all of them will explode, but a few can be fatal, or at least very damaging.

2.1.2 From defects to failures

It takes four stages for a failure to happen. First, the developer introduces a defect; that is, a piece of code that can be the cause of an infection. A defect turns into an infection when this specific piece of code is executed under certain conditions that causes the infection. Once it takes place, the program state differs from the one originally intended by the developer. The propagation of the infection characterizes the third stage of a failure lifecycle. It is important to notice that an infection does not necessarily need to propagate in a continuous fashion. It may be overwritten, masked, or corrected later during program execution. At the fourth stage, the infection causes a failure: an externally observable error in the program behavior caused by an infection in the program state.

Thus, not every defect results in an infection, and not every infection results in a failure. Also, having no failures does not imply having no defects. Dijkstra pointed out that testing can only show the presence of defects, but never their absence (BUXTON; RANDELL et al., 1970, p. 66).

2.1.3 From failures to fixes

Having a failure as a starting point and working it until the defect is fixed is an elaborate process, that demands certain steps to be followed. The first letter of each step forms the TRAFFIC (ZELLER, 2009) acronym, detailed in what follows.

1. **T**rack the problem in the database.
2. **R**eproduce the failure.
3. **A**utomate and simplify the test case.
4. **F**ind possible infection origins.
5. **F**ocus on the most likely origins.
6. **I**solate the infection chain.
7. **C**orrect the defect.

Step 1—*track the problem in the database*—states that all defects must be tracked in a bug tracking tool (such as Bugzilla[2]). By centralizing all problems in a single repository, management can use it as an asset, especially when it is time to decide whether a system is ready for release. This is possible because these tools classify defects by severity and priority. With the aid of a bug tracking tool, one is able to find out which are the main problems and how they can be avoided during development.

Step 2—*reproduce the failure*—should be the first task of any debugging activity, and it must occur exactly as stated in the original problem report. The importance of this step is twofold: (1) if the defect cannot be reproduced, there is only one alternative left—deduce from the source code what might have happened to cause the problem; (2) to ensure that the problem is fixed—if it is not possible to reproduce it, one can never affirm that the problem has actually been fixed.

Step 3—*automate and simplify the test case*. Once a problem has been reproduced, it is time to simplify it. The goal of this step is to find out, for every circumstance associated with the defect, whether it is relevant—something that is absolutely required to make the problem occur—or not. A circumstance is any aspect that may influence the problem. The outcome of this step is a test case containing only the relevant circumstances to recreate the failure. By doing so, it is also possible to track duplicated failure reports registered in a bug tracking tool.

Steps 4, 5 and 6—respectively *find possible infection origins, focus on the most likely origins, and isolate the infection chain*—are the most resource-consuming steps of the process. Understanding the failure is a difficult task because it demands a two-dimensional analysis through spatial and temporal dimensions. The spatial dimension

[2] `http://www.bugzilla.org/`

refers to where the infection takes place in the source code. The temporal dimension determines when the failure happens during the program execution. The examination of space and time are demanding tasks for even simple programs because each state can comprise several variables.

Step 7—*correct the defect*—should be the simplest of all steps; when it is reached, it is expected that the developer has full comprehension of the failure. In order to consider a failure as "fixed", three concerns must be addressed by the developer.

- *Does the failure no longer occur?* One must ensure that the correction makes the failure no longer occur. This requires clarification since there are two ways of fixing a failure: by correcting its root cause or by correcting an infectious state. The former ensures that the defect will not happen again while the latter does not offer the same guarantee.
- *Did the correction introduce new problems?* This can be difficult to verify when there is no automated regression test suite available.
- *Was the same mistake made elsewhere?* One must check for possible defects that may be caused by the same mistake.

2.2 Code coverage

Code coverage—also known as *program spectra* (RENIERIS; REISS, 2003; HARROLD et al., 1998; DICKINSON; LEON; PODGURSKI, 2001)—can be defined as a set of components covered during the execution of test cases (ELBAUM; GABLE; ROTHERMEL, 2001). Components in this context are statements, predicates, def-use associations or function calls (ABREU; ZOETEWEIJ; GEMUND, 2008a).

Therefore, code coverage is associated with dynamic verification of control and/or data flows occurring during the execution of a program. It can be associated with unit (intraprocedural) coverage or integration (interprocedural) coverage. In what follows, we discuss the different types of code coverage.

2.2.1 Unit coverage

Consider the program presented in Figure 3 which determines the maximum element in an array of integers. The lines of code associated with the statements of the

program are presented. The program has a defect localized in line 2.

Typically, a program P is mapped into a flow graph $G(N, E, s, e)$ where N is the set of blocks of statements (nodes) such that once the first statement is executed all statements are executed in sequence, s is the start node, e is the exit node, and E is the set of edges (n',n), such that $n' \neq n$, which represents a possible transfer of control between node n' and node n (CHAIM; ARAUJO, 2013). Figure 4 contains the flow graph obtained from program max; and Figure 3 lists the statements associated with each node.

Line	Statement	Node	Code
1	-	-	int max(int array [], int length)
2	-	1	{
3	1	1	int i = 0;
4	2	1	int max = array[++i]; **//array[i++];**
5	3	2	while(i < length)
6	-	3	{
7	4	3	if(array[i] > max)
8	5	4	max = array[i];
9	6	5	i = i + 1;
10	-	5	}
11	7	6	return max;
12	-	6	}

Figure 3 *– Example program max (CHAIM; ARAUJO, 2013).*

A brief explanation of the different types of code coverage is presented below.

- *Statement coverage* identifies the statements traversed by a single test case or test suite.

- *Node coverage* refers to nodes exercised by test cases. Statement and node coverage are similar. The difference may arise when, for example, conditional assignments are present in a statement since there are more than one node in such statements (SOUZA; CHAIM, 2013). A test set that satisfies the node coverage always satisfies the statement coverage. The opposite is not always true.

- *Predicate coverage* identifies predicate commands traversed by the test case. A predicate is a boolean expression referring to some property present in some point of the program (ZHANG et al., 2009). Predicate commands are associated with a condition verified during an execution, in which a deviation in the control flow can occur, such as in *if* and *while* commands. For example, there are predicates in lines 3 and 4 of program *max*.

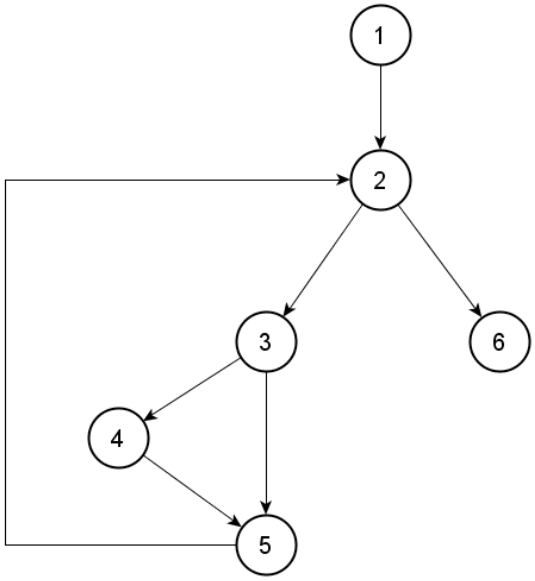


Figure 4 – *Control-flow graph for program max.*

Table 1 – *Test cases for program max.*

T_n	Input data	Expected Outcome	Actual Outcome
t_1	max([1,2,3] , 3)	3	3
t_2	max([5,5,6] , 3)	6	6
t_3	max([2,1,10] , 3)	10	10
t_4	max([4,3,2] , 3)	4	3
t_5	max([4] , 1)	4	error

To illustrate the coverage regarding program *max*, consider Table 1. It presents five test cases (t_1, t_2, t_3, t_4, and $t5$) to test *max*. When executed, test cases t_4 and t_5 fail, while t_1, t_2, and t_3 pass.

Table 2 presents the statement and node coverage for each test case. The bullet associated with each statement, node and test case means that the particular test case exercises during testing these components. Table 3 presents predicate coverage. Note that each predicate has two outcomes: true or false. A test case may execute both outcomes or only one of them.

Coverages based on the program data-flow are also used for debugging purposes (SANTELICES et al., 2009; CHAIM; MALDONADO; JINO, 2004). Definition-use associations (duas) (RAPPS; WEYUKER, 1985), which involves a definition of a variable (an assignment of value) and its subsequent use (a reference to this value), is an example of data-flow coverage. Since our goal is to address the visualization of debugging information obtained from unit and integration control-flow coverages, data-flow coverages are out of the scope

Table 2 – *Statement and node coverage of program max.*

Line	Node	t_1	t_2	t_3	t_4	t_5
3	1	●	●	●	●	●
4	1	●	●	●	●	●
5	2	●	●	●	●	
7	3	●	●	●	●	
8	4	●	●	●		
9	5	●	●	●	●	
11	6	●	●	●	●	
-	-	Pass	Pass	Pass	Fail	Fail

Table 3 – *Predicate coverage of program max.*

Predicate		t_1	t_2	t_3	t_4	t_5
Line	Outcome					
5	true	●	●	●	●	
5	false	●	●	●	●	
7	true	●	●	●		
7	false	●	●	●	●	
-	-	Pass	Pass	Pass	Fail	Fail

of this work.

2.2.2 Integration coverage

The aforementioned coverages are obtained by testing the unit, being a unit a method of a program[3]. Nevertheless, coverages related to integration testing data can also be used for debugging purposes.

A typical integration coverage is *module or call function coverage* which reports the modules (e.g., procedures, functions, or methods) exercised by a test case or suite. In this work, we utilize the *MethodCallPair* (MCP) integration coverage which was proposed by Souza (SOUZA, 2012; SOUZA; CHAIM, 2013) and is described as follows.

The *MethodCallPair* (MCP) integration coverage was developed to collect information about the most suspicious method call sequences. MCP was not devised as a testing requirement. Rather, it was conceived as integration information captured during test suite execution for debugging purposes.

MCP represents pairs of methods exercised during a test suite run. A pair is composed of a *caller* method and a *callee* method. Instead of simply capturing the

[3]The term *method* is utilized to refer to a module (e.g., procedure, function, method) in a program.

method call stack to calculate their suspiciousness, the *rationale* is to highlight methods more often related to other methods in failed executions. Thus, the integration between methods is used to indicate those more likely to contain faults. Figure 5 illustrates MCP coverage information in which the method *caller* of the class *A* invokes the method *callee* of the class *B*.

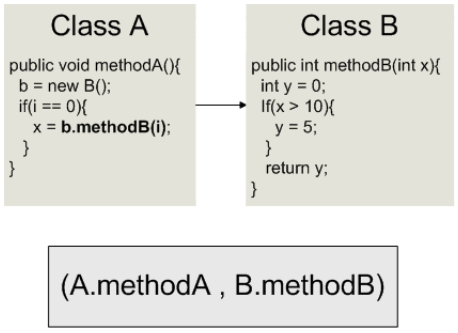


Figure 5 – *MethodCallPair (MCP).*

Several techniques utilize heuristics that are based on the execution frequency of nodes, statements, predicates and MCPs to find out the ones that are more suspicious to contain defects (SOUZA, 2012).

The following section presents further details on how code coverage information is used to support debugging activities, which will provide the foundations for our novel metaphor, called CodeForest.

2.3 Coverage based debugging

Code coverage information is utilized by heuristics to calculate the suspiciousness of components. After assigning a suspiciousness value to these components, a classification list of components in descending order of suspiciousness is generated.

Thus, the heuristics used to elect suspicious excerpts of code are a pivotal issue in coverage based debugging. Several heuristics have been proposed or utilized to indicate components (statements, predicates, def-use associations, methods or MCPs) most likely to contain faults. Ranking heuristics—characterized by identifying suspicious code excerpts from the frequency of executed components in passing and failing test cases—are prevalent. There are ranking heuristics created specifically for fault localization (JONES; HARROLD; STASKO, 2002; WONG et al., 2007), while others have been adapted from areas such as Molecular Biology (ABREU; ZOETEWEIJ; GEMUND, 2007; GONZÁLEZ, 2007).

In what follows, debugging techniques based on code coverage are described.

Firstly, techniques that utilize unit code coverage are presented, then the use of integration coverage in fault localization is discussed.

2.3.1 Unit coverage debugging

The *Tarantula* heuristic (JONES; HARROLD; STASKO, 2002) is one of the first ranking heuristics utilized to assign suspiciousness values to program statements. For each component, Tarantula calculates the frequency that a component is executed in failing test cases and divides it by the frequency that this component is executed in failing and passing test cases.

Tarantula's formula H_T is shown below in Equation 2.1: c_{ef} indicates the number of times that a component (c) is executed (e) in failing (f) test cases, c_{nf} is the number of times that a component is not (n) executed in failing (f) test cases, c_{ep} is the number of times that a component is executed (e) by a passing (p) test case and c_{np} represents the number of times that a component is not (n) executed by passing (p) test cases.

$$H_T = \frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}} + \frac{c_{ep}}{c_{ep}+c_{np}}} \tag{2.1}$$

Table 4 presents the statement and node coverage, the outcome of each test presented in Table 1 and the values of the coefficients utilized to determine the suspiciousness using Tarantula. The suspiciousness value for lines 3 and 4 is 0.5 and for the rest of the lines is 0.33 or 0. Therefore, lines 3 and 4 (i.e., node 1) are the most suspicious statements according to Tarantula. Other heuristics utilize the same coefficients with different formulae to determine the suspiciousness values.

2.3.2 Integration coverage debugging

A limitation of the current ranking heuristics is the lack of guidance to search for faults in larger portions of code since they assign suspiciousness values only to statements. Parnin and Orso (PARNIN; ORSO, 2011) performed experiments with a group of developers using Tarantula and observed that developers take into account their knowledge of the code to search for faults without usually following Tarantula's classification. In this sense, the results suggest that contextual guidance might be helpful to coverage based debugging.

We discuss below two approaches that utilize integration coverage to provide contextual support in the search for bugs.

Table 4 *– Statement and node coverage of program max.*

Line	Statement	Node	t_1	t_2	t_3	t_4	t_5	c_{np}	c_{ep}	c_{nf}	c_{ef}	H_T
1	-	-	•	•	•	•	•	0	3	0	2	0.5
2	-	1	•	•	•	•	•	0	3	0	2	0.5
3	1	1	•	•	•	•	•	0	3	0	2	0.5
4	2	1	•	•	•	•	•	0	3	0	2	0.5
5	3	2	•	•	•	•		0	3	1	1	0.33
6	-	3	•	•	•	•		0	3	1	1	0.33
7	4	3	•	•	•	•		0	3	1	1	0.33
8	5	4	•	•	•			0	3	2	0	0
9	6	5	•	•	•	•		0	3	1	1	0.33
10	-	5	•	•	•	•		0	3	1	1	0.33
11	7	6	•	•	•	•		0	3	1	1	0.33
12	-	6	•	•	•	•		0	3	1	1	0.33
-	-	-	Pass	Pass	Pass	Fail	Fail	-	-	-	-	-

2.3.2.1 Code hierarchy

Souza (SOUZA, 2012) proposed a search heuristic called *code hierarchy* (CH) that attributes suspiciousness values to packages, classes and methods. The CH suspiciousness value is given by a pair (*susp*, *number*) where *susp* is the highest suspiciousness value assigned to a node belonging to the program entity (package, class or method) and *number* is the number of nodes which has that suspiciousness value.

Algorithm 1 shows how the CH value is assigned to classes. It requires that the suspiciousness values of the nodes be previously determined. From now on, the suspiciousness values assigned to nodes are determined by utilizing Tarantula, although any other ranking heuristic (WONG et al., 2007; ABREU; ZOETEWEIJ; GEMUND, 2007; GONZÁLEZ, 2007) can be used as well. The algorithms to calculate CH values to packages and methods are similar.

Packages, classes, and methods are firstly ordered by their *susp* value and then inversely by the *number* value. CH establishes that the *susp* value is the most important factor for fault localization. Thus, if a method has only one node with the highest suspiciousness value, it should be investigated first. In case of a draw, the number of nodes with the same *susp* value decides which is the next entity to be investigated. The smaller the "number", the better. In case of a double draw, the next entity is decided randomly.

In our visual representation of programs, we utilize CH suspiciousness values to determine the positioning and coloring of packages, classes, and methods.

```
   Input: A list of nodes and a list of classes (nodeList, classList)
   Output: A list of classes with CH values (susp, number) assigned
1  foreach class in classList do
2      foreach node in nodeList do
3          if node.class = class then
4              if node.susp > class.susp then
5                  class.susp ← node.susp;
6                  class.number ← 1;
7              else if node.susp = class.susp then
8                  class.number++;
9  return classList;
```

Algorithm 1: Assignment of CH values to classes

2.3.2.2 Creating roadmaps for fault localization

Souza (SOUZA, 2012) proposed to add contextual information to support fault localization by creating roadmaps from the ranking of the MCP coverage. The roadmap, referred to as R-MCP, is a simplified guide to be used when searching for faults. It is determined according to the following steps.

1. Pairs of methods (MCPs) are tracked during the execution of each test case.

2. MCP coverage collected during the test suite execution is used to assign suspiciousness values to each MCP. Any ranking heuristic can be used with this purpose. In the case of using Tarantula, coverage data regarding pairs of methods (MCPs) are the components of the formula presented in Equation 2.1.

3. A list of MCPs sorted by suspiciousness is created. MCPs with the same suspiciousness values are sorted by their order of occurrence in the execution.

4. R-MCP is created by visiting the MCP list from the top ranked elements to the least ranked ones, and from the caller method to the callee method. The caller method and the callee method of each visited pair is verified in this order. If a method is not yet in the roadmap, it is added to it with the same suspiciousness value assigned to the visited pair. Thus, the roadmap contains each method executed in the test suite with its highest suspiciousness in the MCP list.

Consider that an MCP (*A.methodA*,*B.methodB*) is executed in 4 out of 10 test cases that fail and it is executed only in 2 test cases that pass in a total of 90 successful

test cases. Thus, coefficients c_{ef}, c_{nf}, c_{ep}, and c_{np} of equation 2.1 will be, respectively, 4, 6, 2, and 88. As a result, the Tarantula suspiciousness value (H_T) will be 0.941 for MCP (*A.methodA*,*B.methodB*). Once all MCPs have been assigned their H_T, they are sorted in decreasing order; the list of sorted MCPs is then used to create the roadmap. During R-MCP creation, if *A.methodA* is not in the roadmap yet, it is inserted in R-MCP with assigned suspiciouness 0.941. Next, *B.methodB* is checked. If not yet in R-MCP, *B.methodB* is also inserted with suspiciousness 0.941.

The rationale to create R-MCP is based on a developer using a sorted MCP list to search for faults: the developer looks at the top MCP of the list and examines the caller method. If it does not contain the bug, he/she looks at the callee method. If the bug is not discovered by visiting the methods of the top MCP, the search proceeds by inspecting the methods of the following MCPs. When checking an MCP, if a method is seen anew, the developer does not inspect the method again. Instead, he/she skips the method and proceeds with the next not inspected method until finding the bug.

Therefore, R-MCP is composed of each method in the MCP list with its highest suspiciousness value. Methods that appear in more than one pair in the MCP list are included only once in R-MCP. Thus, by using R-MCP, methods are not checked repeatedly. Table 5 presents an example of R-MCP generated for a faulty version of the Apache Commons-Math program (SOUZA, 2012).

Table 5 – *R-MCP roadmap for Commons-Math AE_AK_1 bug.*

method	**suspiciousness**
getCovariances	0.9985037
getAllParameters	0.9985037
getMeasurements	0.9985037
EstimatedParameter	0.997012
getName	0.9960199
solve	0.9955246
buildMessage	0.9955246
MathException	0.9955246
EstimationException	0.9955246
updateJacobian	0.9950298
GaussNewtonEstimator	0.9930556
estimate	0.9930556
SimpleEstimationProblem	0.9930556
getUnboundParameters	0.9930556
addParameter	0.9930556
addMeasurement	0.9930556

The R-MCP's underlying intuition is that the developer will start the exploration of the code by the top ranked methods. When visiting a particular method, he/she will resort to unit coverage debugging to investigate the method. In debugging Commons-Math bug AE_AK_1, the developer would start by the method *getCovariances()* and then move to the next ones until finding the bug site. In failing to do so, the developer will need to resort to another debugging technique.

In most cases, a method is composed of nodes that are classified with different suspiciousness values. The delta suspiciousness (Δ_s) is a strategy proposed to deal with this issue. The idea is to investigate nodes with suspiciousness values slightly lower than the method suspiciousness. The aim is to include the bug in the set of nodes to be investigated without growing excessively the number of nodes to examine.

Thus, a developer will inspect nodes with suspiciousness values equal to or greater than Δ_s. The formula of Δ_s is shown in Equation 2.2. M_s is the method suspiciousness value and p is the percentual value used.

$$\Delta_s = M_s * (1 - p) \tag{2.2}$$

Souza and Chaim (SOUZA; CHAIM, 2013) showed that by using Δ_s with $p = 3\%$ less code is investigated in comparison to a strategy using Tarantula and node coverage. In debugging Commons-Math bug AE_AK_1, only nodes with suspiciousness $0.9985037(1 - 0.03)$, i.e., 0.96854859, will be investigated during the exploration of *getCovariances()* with the delta suspiciousness strategy. By doing so, the authors expect to reduce the number of inspected nodes by visited method.

In Chapter 4, we will present how the suspiciousness values of nodes, methods, and classes are mapped into a visual metaphor. The implementation of the R-MCP roadmap in the plugin for fault localization based on the visual metaphor is discussed in Chapter 5.

In what follows, the concepts regarding information visualization are described.

2.4 Information visualization

The field of visualization is concerned with the mapping of data to pixel values in such a way that meaningful conclusions about the data can be drawn (REINHARD; KHAN; AKYUZ, 2008, p. 397). The essence of this definition includes the analyst, user interfaces,

graphics and data (MYATT; JOHNSON, 2009).

The research field of visualization can be split into two major areas. *Scientific visualization* is primarily concerned with the visualization of multi-dimensional phenomena—architectural, meteorological, medical, biological, etc. *Information visualization* is primarily concerned with the visualization of large-scale collections of non-numerical information, such as large networks, hierarchies, data bases, text, etc. Presenting abstract data, especially very-large scale, still poses a continuing challenge (FRIENDLY; DENIS, 2001).

Statistical graphics and data visualization, despite common belief, are far from being a modern development in statistics. It is possible to trace the art of portraying quantitative information back in time, into the histories of the earliest map-making and visual depiction, and later into thematic cartography, statistics and statistical graphics (FRIENDLY, 2008).

Visual information, when well organized and arranged in a structured way, is capable of elevating our comprehension levels about some information. It is more effective to use colors, shapes and textures—and other visual techniques—to convey information when compared to a text-only approach.

2.4.1 Interface metaphors

According to Lakoff and Johnson, a metaphor is "understanding and experience one kind of thing in terms of another". Metaphors are not only pervasive in language, but they are a fundamental part of our conceptual system of thought and action (LAKOFF; JOHNSON, 1980). They are so important in our lives that there is a proposition that all knowledge is based on metaphors (INDURKHYA, 1994).

The field of Human-Computer Interaction conceives metaphors as cross-domain mappings, with the goal of allowing the transference (or mapping) of knowledge from a source domain (a familiar area of knowledge) to a target domain (unfamiliar area or situation). This enables humans to use specific prior knowledge and experience for understanding and behaving in situations that are novel or unfamiliar (HELANDER; LANDAUER; PRABHU, 1997).

An interface metaphor is a set of user interface visuals, actions and procedures that exploit specific knowledge that users already have of other domains. The purpose is to give users instantaneous knowledge about how to interact with the user interface. It helps establish expectations and encourage predictions about system behavior. A well

known example is the use of the "desktop" metaphor on computers. It portrays an Operating System as objects, tasks and behaviors found in physical office environments. The "Trashcan" found on the Windows Desktop quickly helps users understanding the concept of deletion of files.

The problem with metaphors starts when these comparisons are taken too far. A good example is the first generation of word processors, which heavily relied on the typewriter metaphor. It is a fact that the keyboard of a computer closely resembles the one found in a standard typewriter, and so it seemed like a good metaphor to use. The space key in a typewriter is not a character, it just moves the guide further along the current line. In a word processor, the space is a character, inserted within a text just as any character is inserted. This is the point where the metaphor gets in the way of the user understanding a word processor (DIX, 2004; HELANDER; LANDAUER; PRABHU, 1997).

Other precautions must be taken when dealing with metaphors. Do not assume that a given metaphor is free of cultural bias, i.e, not every metaphor is universal. Also, the number of visual representations provided by the metaphor should be sufficient to cover all objects in the chosen domain. Although possible, it is not advisable to combine representations from different metaphors (DIEHL, 2007).

2.4.2 Visual encoding

The mechanism of vision is intricate and its study is out of the the scope of this research. But in order to better understand the principles that have guided this work, we need to introduce a few concepts regarding the vision mechanism. In fact, we do not see with our eyes; we see with our brains. Our eyes work as complex sensory mechanisms through which light enters. Then, the light is translated into electrical impulses that are passed on to and around our brains. It is in our brain that occurs the process of making sense of what our eyes register actually happens.

Without the filters and limits present in our eyes, the excessive amount of information captured could easily overwhelm our brains. For this reason, our eyes only register what lies within their span of perception. This small portion of what our eyes sense will then become an object of focus. An even smallest fraction of that portion will capture our attention or conscious thought (WANDELL, 1995).

One of the several stages that happens in our brain when performing the vision process is called "preattentive processing". It is one of the early stages of visual per-

ception that rapidly occurs below the level of consciousness. This process is oriented to detect a specific set of visual attributes. A series of experiments conducted on this subject (TREISMAN; GORMICAN, 1988) studied the reaction time of an individual searching for a particular shape (called target) when presented in a set of other shapes (called distracters). The main finding was that, for certain combinations of targets and distracters, the time to find the target *did not* depend on the number of distracters, suggesting that this processing happens in a parallel and automatic fashion.

In his book, Ware (WARE, 2008) suggests seventeen "preattentive" attributes. The most relevant ones were narrowed down in (FEW, 2006), and are shown in Figure 6. These attributes are perceived within 200 milliseconds, i.e., the time interval it takes before the eye reacts and moves (HEALEY; BOOTH; ENNS, 1996).

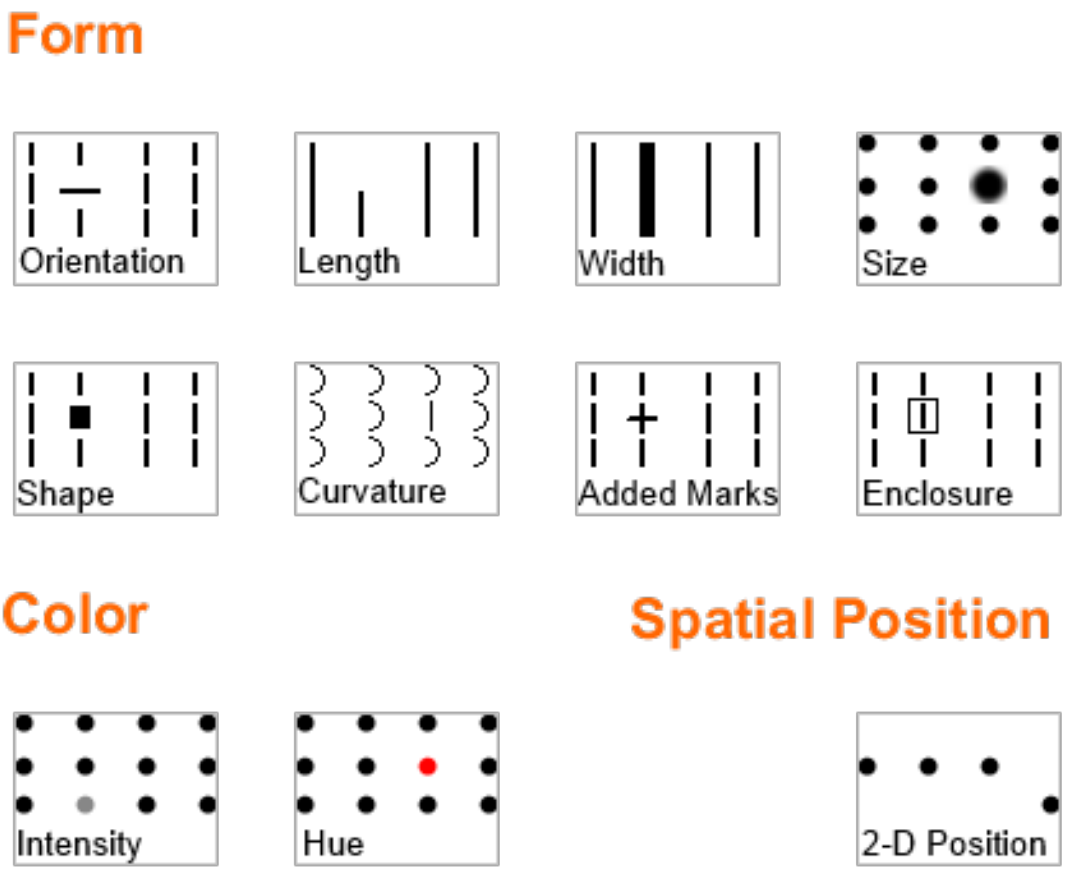


Figure 6 – *Interpretation of Few's preattentive attributes(FEW, 2006)*

After the stage of feature extraction is done, the stage of pattern perception takes place. At this point, the brain starts the segmentation of the visual world into distinct regions to discover the structure of objects and the connections between them. At the

third and final stage, the brain does attentive processes to perform the analytical task at hand.

With this information in mind, we believe that an effective way of conveying information is the one that exploits the first stage of this mechanism, by letting the preattentive processing do some thinking on our behalf. Although it is not a straightforward process due to the fact that the effectiveness of these attributes strongly varies based on the desired data type to encode, there are some "rules of thumb" that can be applied (MACKINLAY, 1986). Table 6 presents a ranking showing the effectiveness as a function of the data type. It is important to notice that picking the most effective encodings relies heavily on the task at hand, so even the encoding rankings should be seen in that light. It is also possible to encode a data type within more than one visual channel, in order to help perform the task more effectively (INC., 2012).

Table 6 – *Mackinlay's rankings (MACKINLAY, 1986)*

Quantitative	**Ordinal**	**Nominal**
Position	Position	Position
Length	Density	Hue
Angle	Saturation	Texture
Slope	Hue	Connection
Area	Texture	Containment
Volume	Connection	Density
Density	Containment	Saturation
Saturation	Length	Shape
Hue	Angle	Length
Texture	Slope	Angle
Connection	Area	Slope
Containment	Volume	Area
Shape	Shape	Volume

A domain set is *quantitative* when it is a range, such as $[1, 2, 3, 4, 5]$; it is *ordinal* when it is an ordered tuple, such as $\langle Monday, Tuesday, Wednesday \rangle$; and it is *nominal* when there is the absence of a relation of order among elements, such as $\{Marcos, Higor, Danilo\}$. Figure 7 presents a comparison between two quantitative pieces of data: the American national average monthly gasoline retail price and the monthly residential electricity price (ADMINISTRATION, 2012). The figure makes use of the following Mackinlay attributes: position, angle, slope, area, color hue, connection and containment.

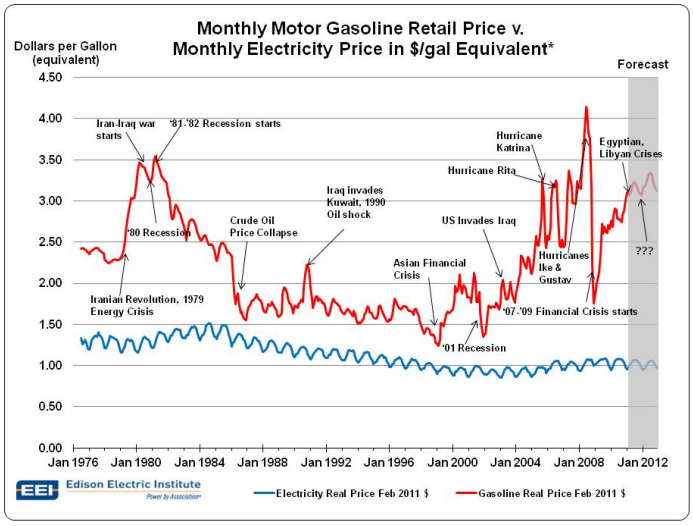


Figure 7 – *National average monthly gasoline retail price versus monthly residential electricity price (ADMINISTRATION, 2012)*

Figure 8 presents the Saudi Arabian Communications and Information Technology Commission organizational structure, as an example of a ordinal set. The purpose of a organizational chart is to show the structure of an organization and the relationships and relative ranks of its parts and position/jobs. There is a clear distinction of which position is above or under or equivalent to which, and who is the head of the organization. The figure makes use of the following Mackinlay attributes: position, color hue, connection, containment, length, area, and shape.

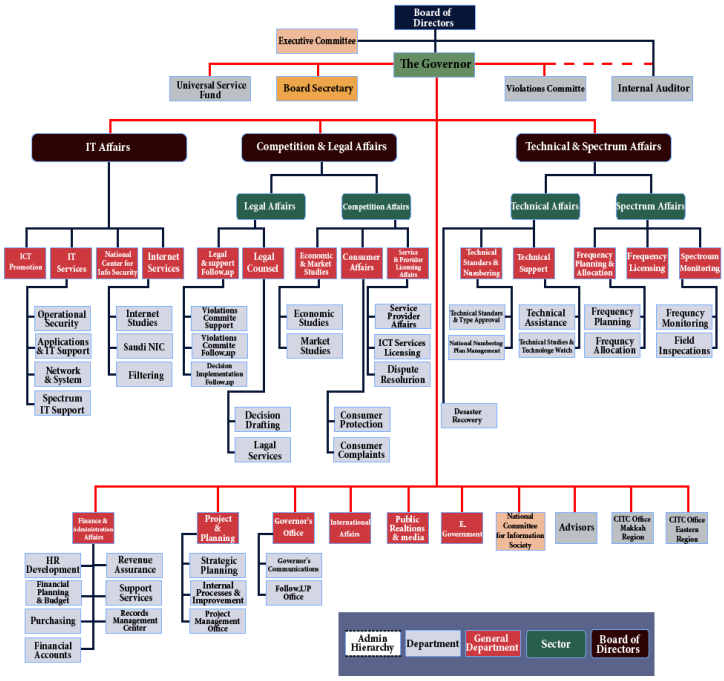


Figure 8 – *CITC organizational structure (COMMUNICATIONS; COMMISSION, 2013)*

Lastly, Figure 9 displays the 2012 american election results between Barack Obama and Mitt Romney, with Obama being declared victor by 332 votes to 206. This is a nominal set because there is no sense of ordering here, with two people contending to be elected. It is possible to see the US national territory and its states (another nominal set) with the number of votes obtained by the winning candidate. The figure makes use of the following Mackinlay attributes: position, color hue, connection, containment, shape, length, and area.

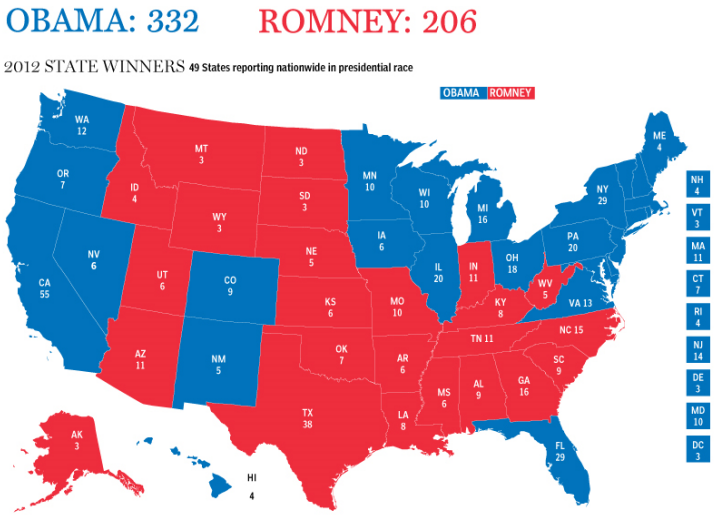


Figure 9 – *U.S. 2012 election results (JOHNSON; RIVAIT; BARR, 2012)*

2.4.3 Software visualization

Considering software visualization as a discipline only concerned with the visualization of algorithms and programs is a too-narrow definition. It is necessary to widen it in a way where this term means *the visualization of artifacts related to software and its development process*. These artifacts may not only be program code, but may also include requirements and design documentation, changes in the source code, bug reports and, in the scope of this research, coverage based debugging information, explained in Section 2.3.

In short, software visualization's objective is visualizing the structure, behavior, and evolution of software (DIEHL, 2007). This field of research is considered a specialization of information visualization—since algorithms, files and lines of code in software systems are a kind of information—whose goal is to help to comprehend software systems and to improve the productivity of the software development process.

To achieve the previously stated objective, it is important to follow at least one

of the following principles.

Principle of interactive exploration: This principle, also called The Information Seeking Mantra (SHNEIDERMAN, 1996), states that in order to explore the data, interactive visualizations should allow the user to first get an overview, and then zoom and filter, getting the details on demand.

Principle of Focus + Context: This principle states that a detailed visualization of some part of the information—the focus—is embedded within a visualization of the context. What it means is that techniques that cope with this principle must provide both an overview and detail at the same time.

2.5 Final remarks

This chapter presented the fundamental concepts and techniques that form the foundation for this research, providing the necessary background knowledge on testing (Section 2.1), code coverage (Section 2.2), coverage based debugging (Section 2.3), and information visualization (Section 2.4). A more in-depth discussion on software visualization will be presented in Chapter 3.

Chapter 3

Related Work

The goal of this chapter is to characterize the state-of-the-art regarding methods and techniques of Information Visualization that can be or are already being applied to Software Visualization, with special emphasis on visualization of debugging information.

3.1 Research methodology

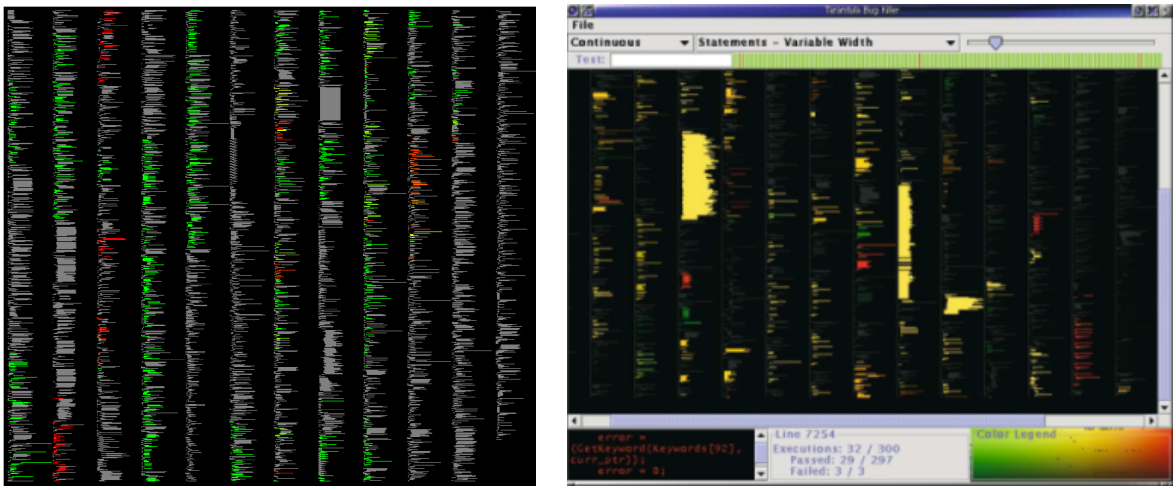
In order to build a foundation for this literature review, the first stage consisted of a selection of seminal papers comprehending the areas of *Information visualization* (InfoVis) and *Debugging Information Visualization*. Then, the next task was to list all research derived from those papers. This was done with the aid of important scientific knowledge bases available on the Internet: the *IEEEXplore Digital Library* (IEEE), the *ACM Digital Library* (ACM), and the *Google Scholar*.

The third stage consisted of reading the abstract found in the articles, checking for its adherence to the axis of this research. Next, those that offered the highest contributions to this research were chosen for deeper analysis and extraction of its content. At last, the remaining articles were divided in two distinct groups, one with two-dimensional visualization and the other containing three-dimensional visualization articles.

The following sections present the results found in this literature review, a table summarizing the chosen articles and the final remarks of this chapter.

3.2 Two-dimensional visualization

Three-dimensional technology was not popular until the end of 1990's, mainly because of low-performance graphical processing units (GPUs) available at the time, which were unable to adequately display 3D graphics. For this reason, most visualization schemes presented so far are limited to two-dimensional worlds. Although it would be incorrect to say that hardware limitation is the sole reason that justifies this popularity. Two-dimensional worlds poses almost no challenge for the regular user, because what



(a) Seesoft (EICK; STEFFEN; JR, 1992) (b) Tarantula (JONES; HARROLD; STASKO, 2002)

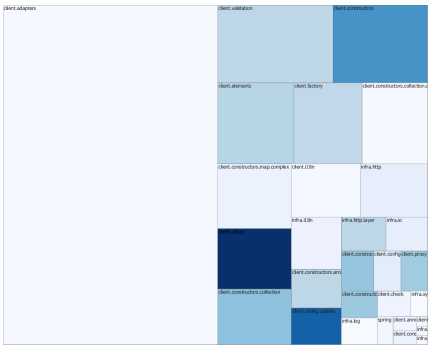
Figure 10 – *Seesoft-like visualizations*

probably is the second most widespread device of interaction known by computer-users, the mouse (perhaps the first is the keyboard) works on 2D.

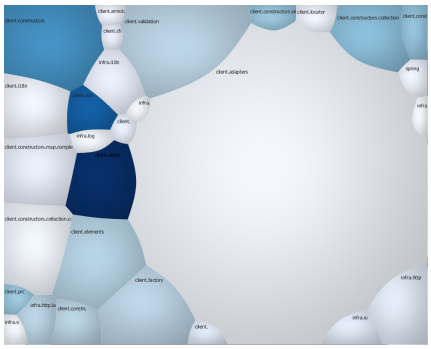
It is safe to say that no research on Software Visualization is complete if it does not mention *Seesoft* (Figure 10(a)). This system was developed at AT&T Bell Laboratories and its purpose was to visualize changes and metrics related to evolving large—up to 60.000 lines of code, split into several screens—complex software systems. The rationale is that every line of code is mapped into a short, horizontal line of pixels. The color of each row indicates a statistic of interest, i.e., blue lines indicate least recently changed lines of code, whereas red lines indicate the most recently changed. Jones et al. (JONES; HARROLD; STASKO, 2002) and Orso et al. (ORSO et al., 2004) utilized the same implementation to display other information, such as coverage based debugging information. In this case, red lines pinpoint locations in the source code with the highest probability of containing a defect (Figure 10(b)).

When it comes to displaying hierarchically structured information, one solution at hand is the treemap technique, which is based on a 2D space-filling approach. Each node of a directed graph is a rectangle whose area is proportional to some attribute, such as node size (Figure 11(a)). The motivation for this work was to have a better representation of storage space occupied by directories and files on a hard disk. A manifold of research papers successfully adapted treemaps to represent different characteristics of source code structure (Figure 11(b)).

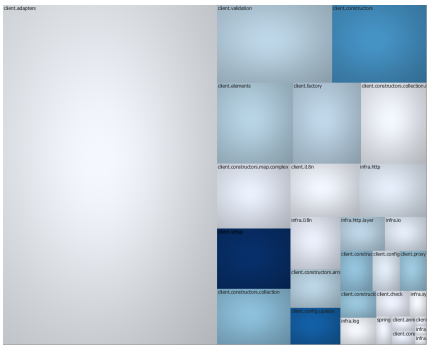
The goal of displaying coverage based debugging information was also explored by the *GZoltar* tool (CAMPOS et al., 2012), which offers two types of diagrams: adjacency



(a) Treemaps (SHNEIDERMAN, 1992)

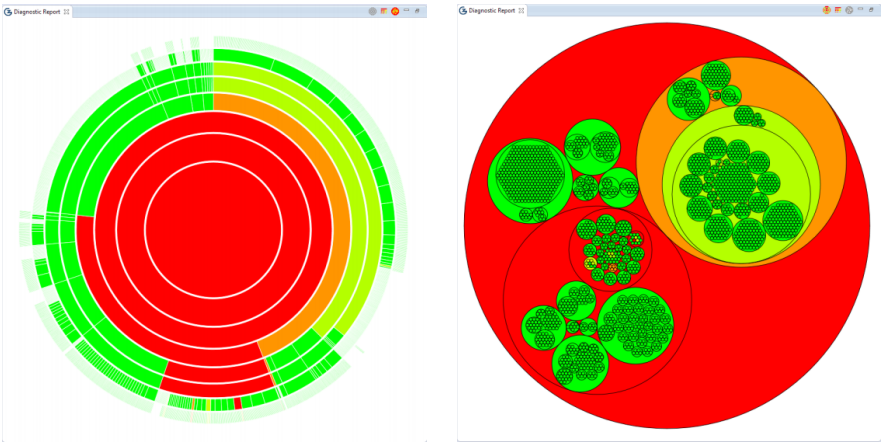


(b) Voronoi treemaps (BALZER; DEUSSEN; LEWERENTZ, 2005)



(c) Cushion treemaps (WIJK; WETERING, 1999)

Figure 11 – *ReConf as Treemap-based visualizations*



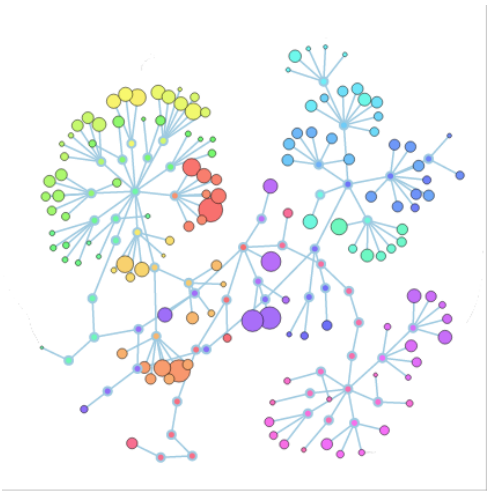
(a) Sunburst (b) Bubble Hierarchy

Figure 12 *– GZoltar (CAMPOS et al., 2012) visualizations*

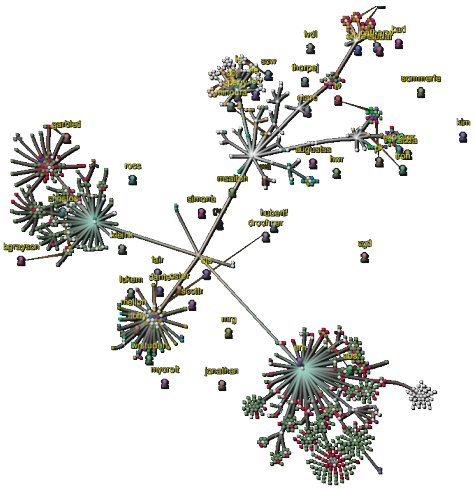
and enclosure. The first is implemented by Sunburst (Figure 12(a)); the latter by the Bubble Hierarchy visualization (Figure 12(b)). Both visualizations utilize the space-filling algorithm from the treemap technique. The tool allows developers to zoom areas of interest, from packages to lines of code. The structure of the visualization remains the same; the greater the zoom the finer is the element being displayed.

CodeFlowers (Figure 13(a)) and *Code clones* (Figure 13(c)) have very similar representations for radically different objectives. Both works represent source code files as discs, whose size is proportional to the number of lines of code found in each file. *CodeFlowers* displays source repositories as an interactive tree where each disc represents a file, with a radius proportional to the number of lines of code. The root node (main directory) is linked to child nodes (subdirectories) and source code files. Every directory that spans to other directories and/or other files creates a hub (ZANINOTTO, 2013). *Code clones* follows the same strategy, mapping source code files to discs. The tool connects files with a link when the similarity between them (clones) exceeds the threshold defined by the user (YOSHIMURA; MIBE, 2012). Though it is not possible to find any details of how a color is picked, *Code clones* seems to choose it according to the the percentage of similarity found among different files. *CodeFlowers* appears to consider the number of lines of code: the greater it is, the darker is the color. *Gource* (CAUDWELL, 2010) uses the same visual concept—discs and branches—to map the evolution of a software system to a graphic representation, showing the evolution in the form of an animation (Figure 13(b)). This approach contrasts with the previous interfaces, whose focus is the latest snapshot of a project.

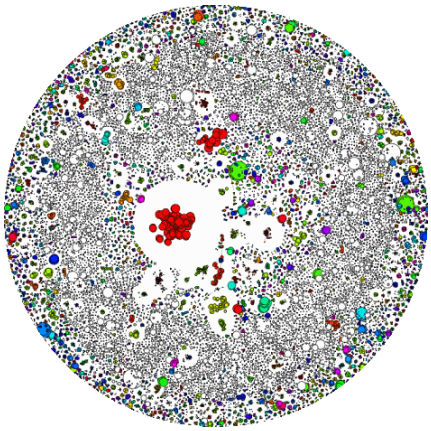
Code Bubbles (Figure 14) proposed a new way of how users should interact with



(a) CodeFlowers (ZANINOTTO, 2013)



(b) Gource (CAUDWELL, 2010)



(c) Code clones (YOSHIMURA; MIBE, 2012)

Figure 13 – *Graph-based visualizations*

an Integrated Development Environment (*IDE*). It stands out because it shifts the traditional approach—adopted by most of IDEs— of focusing on files, to focus on methods. The intent is to create something that would enhance software development tasks such as complex debugging, testing, and collaboration amongst developers. The rationale is that when in a debugging session, the user navigates through bubbles—small windows presenting specific content, e.g., the body of a method being entered—that the developer can create or the ones automatically created by the interface. The tool offers specialized kinds of bubbles to display notes, methods, stack traces and the current value of variables during a live debugging session. Another valuable contribution found in *Code Bubbles* is a feature called *LogBook*, which provides an annotated and searchable history accumulated during software development. The user can add notes, screenshots and other documents to the log. This module is also capable of automatically tracking important events and adding them to the log.

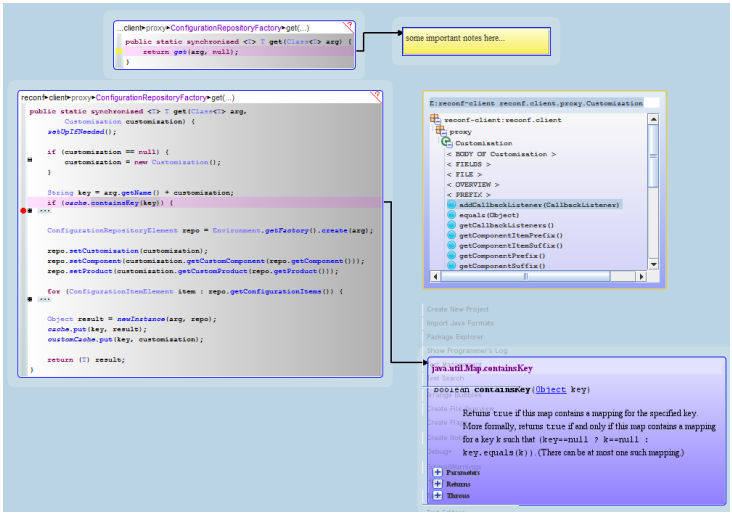
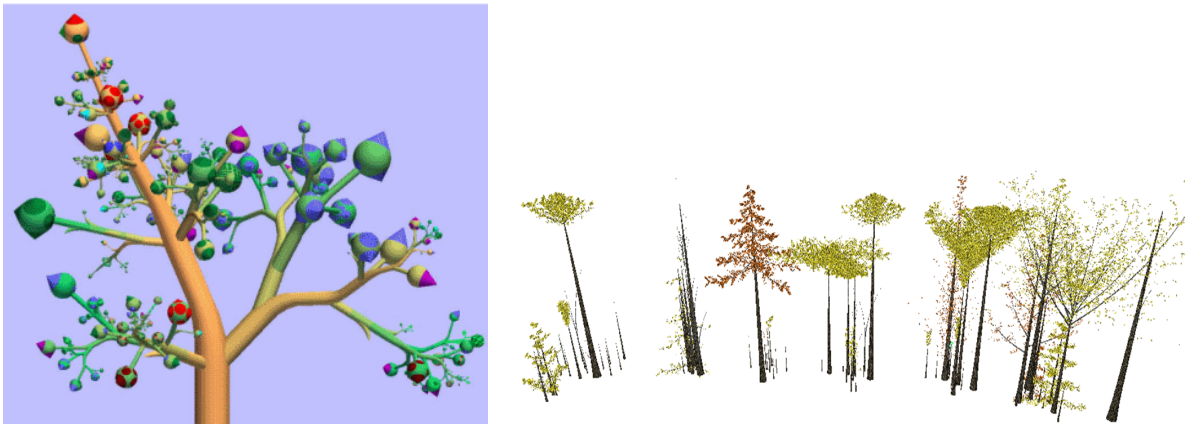


Figure 14 – *Code Bubbles (REISS; BOTT; LAVIOLA, 2012)*

3.3 Three-dimensional visualization

Three-dimensional (3D) visualization techniques have been explored in a variety of software characteristics, with special emphasis on source code metrics.

Despite the abundance of source code oriented visualizations, we prefer to start our review on this subject with a notable exception. The work of Kleiberg et al. (KLEIBERG; WETERING; WIJK, 2001) describes a technique for 3D visualization of huge hierarchies by



(a) Botanical trees (KLEIBERG; WETERING; WIJK, 2001)

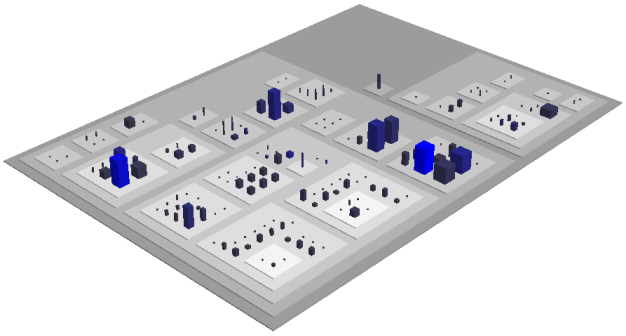
(b) CodeTrees (ERRA; SCANNIELLO, 2012)

Figure 15 *– Botanical tree-based visualizations*

using botanical trees (Figure 15(a)). Their research presents a prototype capable of rendering the whole Linux filesystem as a tree. Initial versions of the tool focused on a faithful representation of a tree with all its elements. However, the large number of leaves (the representation of files) added too much noise to the visualization. In order to avoid the cluttering effect, the authors had to replace the leaf element with a new one, but still keeping the metaphor faithful to the original concept. The concept of a fruit was born, an agglomeration of leaves, implemented as spheres in which leaves are represented as smallest spheres studded on a large sphere, called the phi-ball.

A tree is capable of transmitting an unmistakable notion of hierarchy due to the way it is naturally structured. Leaves are attached to branches which are analogously attached to the trunk. One of the main advantages of presenting software this way is that trees have an instant match of almost all concepts embodied in an object oriented language like Java. Erra and Scanniello (ERRA; SCANNIELLO, 2012) built a tool that maps classes to trees, methods to branches, and lines of code (LOCs) to leaves (Figure 15(b)). The actual number of LOCs (without comments) of a class defines the height of the corresponding tree, while all methods, disregarding its modifiers (private, public, default, and protected) have a matching branch in the trunk. Even though the prototype takes into consideration lines of code, all leaves are painted with the same color (defined by the ratio between actual LOCs and comments). The purpose is to give developers a large scale vision of the software, serving as a guide for spotting patterns in the general structure, not in the code itself.

Several authors have explored the idea of representing software metrics as cities of code (WETTEL; LANZA, 2007; KNIGHT; MUNRO, 2000; PANAS; BERRIGAN; GRUNDY,



(a) CodeCity (WETTEL; LANZA, 2007)

(b) 3D city (KNIGHT; MUNRO, 2000)

(c) Software world (PANAS; BERRIGAN; GRUNDY, 2003)

Figure 16 – *City-like visualizations*

2003). These methods have a commonality of representing software components, in which packages are mapped either as neighbourhoods or whole cities, and classes and files are usually mapped into buildings. It is a common strategy amongst these works to take into consideration the number of inner attributes of a class to determine the building's height (Figures 16(a), 16(b), and 16(c)).

This metaphor transmits a clear notion of hierarchy between its elements, although it is naturally limited on two levels, i.e., neighbourhoods (or cities) and buildings. It is an interesting approach when the developer is solely focused on understanding how coarse grained building blocks (classes and packages) relate to each other. We believe that its main drawback comes from the fact that it is incapable of giving users an idea of what is going on "under the hood", e.g., the number of lines of code found in methods.

Panas et al. (PANAS; BERRIGAN; GRUNDY, 2003) took the city of code a step further, by merging information gathered from version control systems (VCS) with the city structure, in a way to represent other characteristics such as frequently modified files (buildings on fire) or deprecated classes (old buildings). This virtual world is also capable of displaying information about the control flow between classes. If static analysis points

out that method calls happen in a unidirectional way (only class A calls class B), it is rendered as a river. If, otherwise, the control flow is bidirectional, then these two cities are connected by a highway. In order to give users a more immersive experience, other aesthetic-only elements have been added; for example, street and traffic lights.

Graham et al. (GRAHAM; YANG; BERRIGAN, 2004) proposed the use of solar systems instead of cities. The components of the view are planets (Java classes or interfaces), orbits (the inheritance level within the package) and connectors (coupling between classes). The number of LOCs found within classes is responsible for the resulting size of planets. According to the authors, the city metaphor reduces the ability to present complex relationships, such as inheritance among classes. Another drawback is that the city itself is a complex entity to display, with lots of details. The paper claims that a simpler metaphor, with less aestethic details can be as or more effective way of transmitting information than cities.

3.4 Discussion

Tables 7, 8 and 9 presents a summary from the discussion found in Sections 3.2 and 3.3. The structure of the tables follow the recommendations preconized by (LARAMEE, 2011), which advocates that, when extracting the data from a visualization research paper, one should make a clear distinction between concept and implementation. Concept should be understood as "what, conceptually, are the authors trying to achieve? What's the research's goal?"(LARAMEE, 2011, p. 79). On the other hand, the implementation refers to the realization of the concept, the process of translating a goal into a means to achieve it. For example, a pencil is the implementation of a much more abstract concept, which is a writing utensil that allows a person to communicate with other by drawing symbols.

Besides the separation of concept and implementation, each table also includes the aspect approached by the research. Software visualization is concerned with one or more of the following aspects of software: structure, behavior, and evolution. *Structure* refers to the static parts and relations of the system, which includes program code, data structures, static call graph[1], and the organization of the program into modules. *Behavior* refers to the sequence of program states which happen during the execution of the program. *Evolution* refers to the development process of the software system, with focus on the

[1] A call graph is a directed graph that represents calling relationships between functions in a computer program. A static call graph approximates the representation of every possible run of the program. These can include call relationships that would never occur in actual runs of the program (GROVE et al., 1997; RYDER, 1979)

changes that it goes through during construction and maintenance phases (DIEHL, 2007).

Additionally, the intended target audience was also extracted from research papers.

Table 7 *– Related Work Summary - I*

Concept	Implementation	Aspect	Dim	Audience	Reference	Figure
Display test data to aid bug detection	*Seesoft* (EICK; STEFFEN; JR, 1992) to display coverage based debugging information generated by *TARANTULA* heuristics	Structure, Behavior	2D	developers	(JONES; HARROLD; STASKO, 2002)	10(b)
	Sunburst and Bubble Hierarchy to display information generated by the *Ochiai* (ABREU; ZOETEWEIJ; GEMUND, 2006) Heuristics	Structure, Behavior	2D	developers	(CAMPOS et al., 2012)	12(a), 12(b)
Display software structure	Every directory is a hub and and every file is a disc, capable of representing two hierarchical levels. Discs (files) and branches (directories)	Structure	2D	developers	(ZANINOTTO, 2013)	13(a)
Display code clones	Every node in an edgeless graph is a disc, capable of representing two hierarchical levels. Small discs (code clones) and big discs (systems)	Structure	2D	developers, stakeholders, management staff	(YOSHIMURA; MIBE, 2012)	13(c)
Display software execution runtime data	*Seesoft* (EICK; STEFFEN; JR, 1992) to display lines of code and treemap (SHNEIDERMAN, 1992) to display the system as a whole	Structure, Behavior	2D	developers, system administrators	(ORSO et al., 2004)	-

Table 8 *– Related Work Summary - II*

Concept	Implementation	Aspect	Dim	Audience	Reference	Figure
Display software metrics to aid systems comprehension	Voronoi treemaps with tree hierarchical levels: macro-cells (directories), cells (files) and micro-cells (lines of code)	Structure	2D	developers	(BALZER; DEUSSEN; LEWERENTZ, 2005)	11(b)
	Source code as solar systems with two hierarchical levels: suns (directories), and planets (files)	Structure	3D	developers, management staff	(GRAHAM; YANG; BERRIGAN, 2004)	-
	Botanical code trees with three hierarchical levels: trees (files), branches (functions), and leaves (lines of code)	Structure	3D	developers	(ERRA; SCANNIELLO, 2012)	15(b)
	Code cities with two hierarchical levels: neighbourhoods (directories), and buildings (files)	Structure	3D	developers	(WETTEL; LANZA, 2007; KNIGHT; MUNRO, 2000)	16(a), 16(b)
		Structure, Evolution	3D	management staff, stakeholders	(PANAS; BERRIGAN; GRUNDY, 2003)	16(c)

Table 9 *– Related Work Summary - III*

Concept	Implementation	Aspect	Dim	Audience	Reference	Figure
Display source code evolution	Every line of code is displayed as a one-pixel-thin line. Color and hue of every line vary according to the frequency of change	Structure, Evolution	2D	developers	(EICK; STEFFEN; JR, 1992)	10(a)
	A dynamic tree of the active directories generated by a force-directed tree layout algorithm. Users contributing to the project send out beans colored to indicate a kind of action being performed on files (discs)	Structure, Evolution	2D	developers	(CAUDWELL, 2010)	13(b)
Enhance software development tasks	Usage of bubbles to display snippets of code, execution stacks, errors, etc.	Structure, Evolution	2D	developers	(REISS; BOTT; LAVIOLA, 2012)	14
Display huge hierarchies	Botanical directory trees with n hierarchical levels: branches (directories) and fruits (files)	Structure	3D	general	(KLEIBERG; WETERING; WIJK, 2001)	15(a)
	Cushion treemaps with n hierarchical levels	Structure	2D	general	(WIJK; WETERING, 1999)	11(c)

3.5 Final remarks

From our results, it is possible to observe that there is a plethora of research papers dealing with the problem of visualizing software related metrics (such as the number of lines of code, methods, classes, etc.) and relationships, whilst the field of research focused on Debugging Information Visualization remains largely unexplored. This research aims at shortening this gap by providing a novel metaphor and an implementation of it, described in details in Chapter 4.

Chapter 4

CodeForest

This chapter presents a novel metaphor to support coverage based debugging visualization. It also provides details on the realization of the metaphor, as well as on the mechanisms made available to aid developers to narrow their search for faults down to the most suspicious elements (class, method, and node) of the system under analysis.

4.1 Metaphor

The CodeForest metaphor is composed of three elements. Every class is represented as a cactus. A branch is drawn in the cactus for every method within a class. Thorns are associated with nodes. We represent nodes instead of statements (or lines of code) because all statements of a node have the same suspiciousness value since they are executed in sequencial order.

The positioning, coloring, sizing and the thickness of the forest's cacti are determined by their suspiciousness values. First, the cactus having the lowest suspiciousness rank is placed in the upper left corner of the terrain. Then, the element having the second lowest rank is placed to the right side of the previous cactus, and so on. When there is no more space available at the current row, the cactus to be inserted is placed in the left corner just below the previous row. This process goes on until there is no more cacti to be placed in the terrain.

Figure 17(a) illustrates this rationale with the lowest ranked cactus painted in green on the upper left corner; a middle-ranked cactus painted in yellow positioned in the middle; and the highest ranked cactus painted in red placed in the lower right corner—the inspiration for this grid-layout comes from Figure 18, available in (TUFTE; GRAVES-MORRIS, 1983)[p. 108]. The intuition being that these are the system's most likely classes in which the defect might be found. In addition, in order to decrease the developer's distraction levels to a minimum, the higher the rank, the thicker and taller the cactus is. By doing so, the intent is to give as much as possible attention to the first rows where the highest ranked cacti are located. The mapping between color and suspiciousness also

applies to thorns whose colors varies from red (representing the highest ranked node) to green (representing the lowest ranked node).

This rationale is not applied to branches since painting them with colors different from the color (ranking) of the cactus might confuse the developer. Therefore, in order to keep the metaphor informative without turning it into a gathering of colors, a second mapping adds "weight" to a branch. Thus, the highest ranked branches would have a greater weight than lowest ranked branches, with the former positioned closer to the ground, in opposition to the latter. Figure 17(b) illustrates the rationale applied for both thorns and branches regarding their positioning.

The next section describes the realization of the metaphor by presenting in details the strategies employed to achieve the objectives described in Chapter 1.

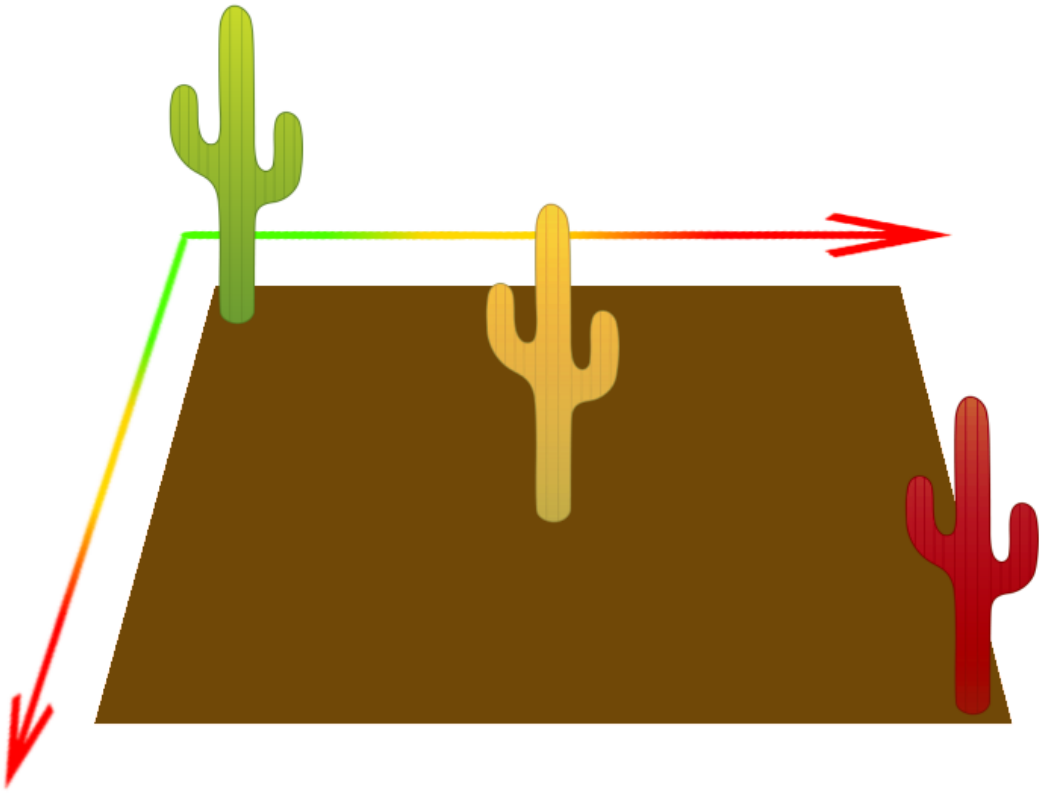
4.2 Metaphor realization

In the previous section, CodeForest's underlying intuition was presented. In this section, we present how coverage debugging information associated with classes, methods, and nodes is mapped into a visual representation of a forest of cacti. The forest elements positioning, coloring, and sizing and their relation to suspiciousness are described next.

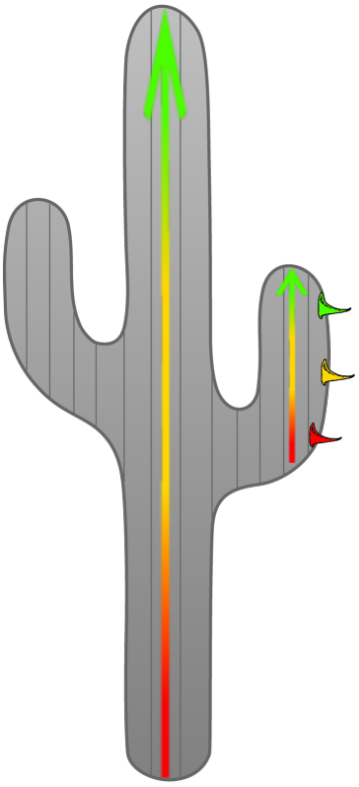
4.2.1 Positioning the elements

The forest is internally represented as a grid where the position of a cactus is determined by its CH suspiciousness value. The first task is to determine the width of the terrain in which the cacti will be positioned. Algorithm 2 demonstrates the steps taken to calculate the ideal width. Briefly, the algorithm calculates the linear width occupied by a cactus by looking for the widest branch on each size (*maxLeftBranchWidth* and *maxRightBranchWidth* functions). Next, the number of cacti plus one is multiplied by the largest trunk radius found (*findMaxRadius* function). The obtained value is added to the summation of the branch size. The ideal terrain width is the result of the division between the sum of all previous values and the square root of the number of cacti in the forest.

As soon as the terrain's width is defined, the cacti's positioning, detailed in Algorithm 3, follows. Cacti are added to a row until the current row gets full (lines 6 to 8), i.e., adding another cactus would cause it to "fall off" the terrain. When this situation



(a) CodeForest



(b) Cactus

Figure 17 – *Proposed metaphors*

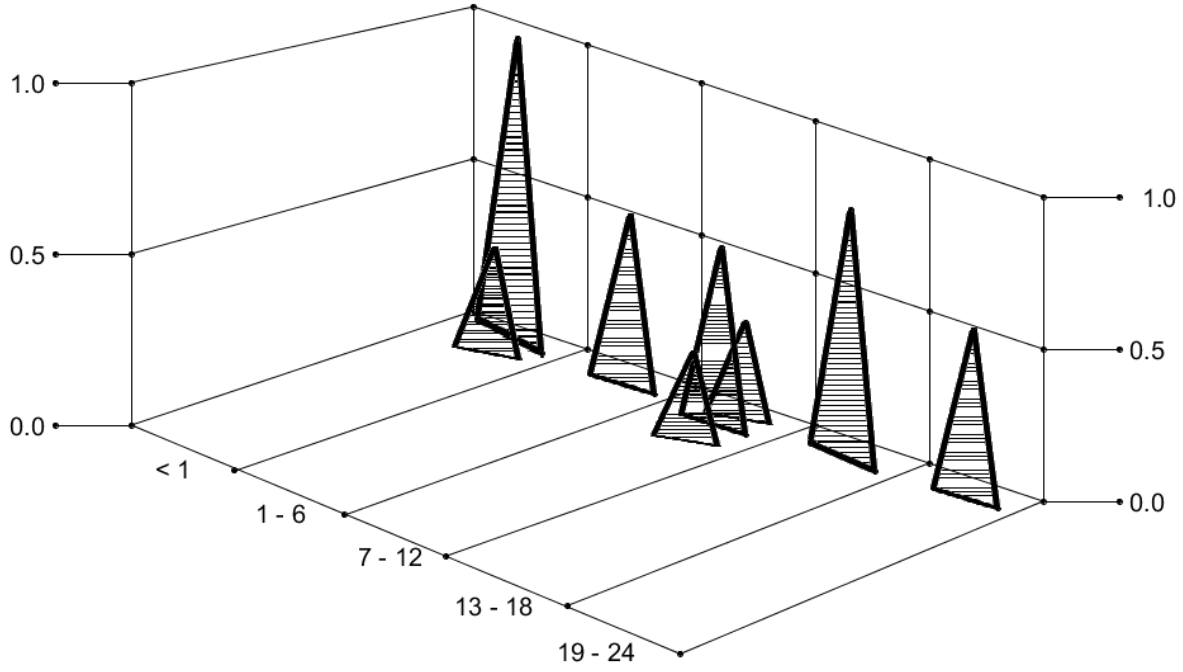


Figure 18 – *Interpretation of Tufte's serial echocardiographic assessments of the severity of regurgitation in the pulmonary autograft in patients (TUFTE; GRAVES-MORRIS, 1983)*

happens, the current cactus is promoted to a newly created row. This algorithm goes on until there are no more cacti to be added. Then, the free space of every row is divided equally among cacti. Thus, when a single cactus occupies an entire row, it is located right in the middle of it, instead of being positioned in the rightmost position. This particular situation can be observed in the aerial view of the Ant[1] program's forest presented in Figure 19. The cactus with the highest rank is located right in the middle of the front row. It is also possible to notice how cacti of different sizes and heights fill different portions of the terrain.

Cacti are positioned in each row according to the description provided in Section 4.1. Each row is populated from the back to the front (lowest to the highest rank); in a row, cacti are ordered from from right to left (highest to the lowest rank). The goal is to focus the programer's attention onto the first rows where the highest ranked cacti are positioned. Thus, these are the classes most likely to contain faults. Figure 20 shows the XML-security[2] program represented as a forest of cacti. In this figure, the most suspicious class is the one up front at far right, indicated by the blue circle.

[1]Ant is the Apache project tool for building applications, especially Java applications (`http://http://ant.apache.org/`).

[2]XML-security is a component library implementing XML signature and encryption standards, supplied by the XML subproject of the open source Apache project (`http://santuario.apache.org/`).

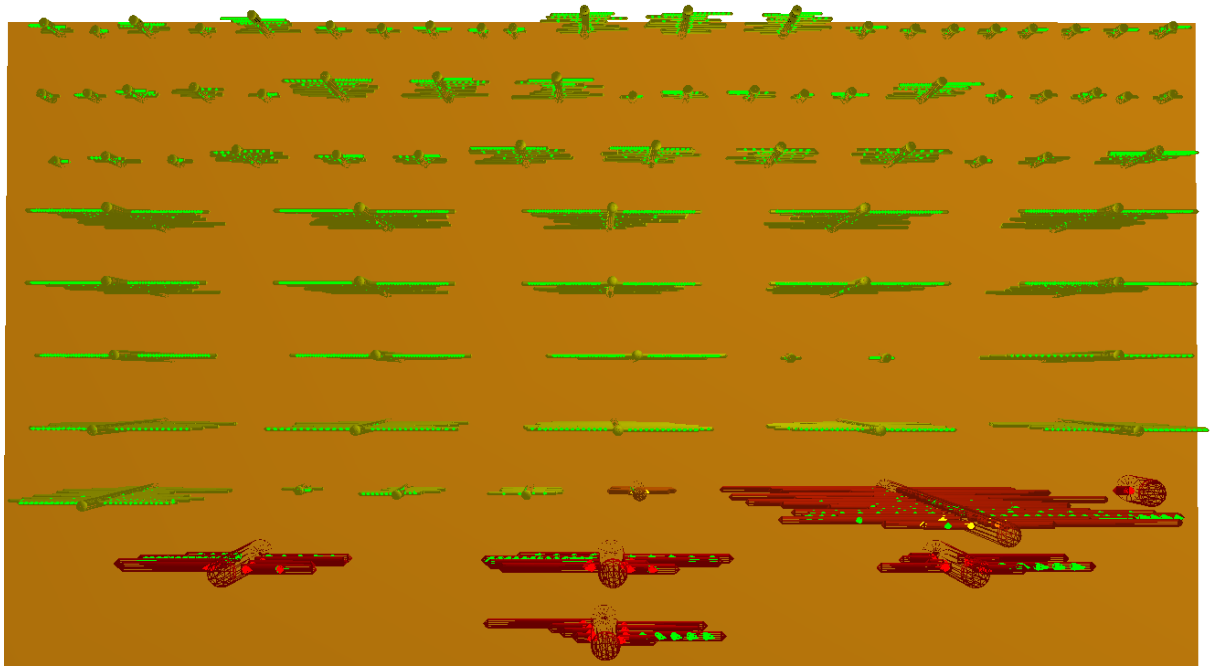


Figure 19 – *Aerial view of Ant's forest of cacti (wireframe).*

Data: An ordered array of records containing CH data (*susp*, *number*) – *classDataList*
Result: Grid width dimension

```
1  begin
2      sum ← 0;
3      count ← 0;
4      Cactus[ ] cacti;
5      foreach classData in classDataList do
6          Cactus cact ← new Cactus(classData);
7          sum ← sum + cact.maxLeftBranchWidth + cact.maxRightBranchWidth;
8          cacti[count] ← cact;
9          count ← count + 1
10     sum ← sum + ((count + 1) * findMaxRadius(cacti));
11     return sum/√count;
```

Algorithm 2: Ideal grid width calculation

```
Data: An ordered array of cactusData – cactiList, ideal grid width – width
Result: Each cactus gets its position on the grid
1  begin
2      i ← 0;
3      grid ← new Grid();
4      grid.row[i] ← new Row();
5      foreach cactusData in cactiList do
6          gap ← width − grid.row[i].currentWidth;
7          if (cactusData.width ≤ gap) then
8              grid.row[i].add(cactusData);
9          else
10             i ← i + 1;
11             grid.row[i] ← new Row();
12             grid.row[i].add(cactusData);
13         grid.row[i].currentWidth ← grid.row[i].currentWidth + cactusData.width;
```

Algorithm 3: Cactus positioning

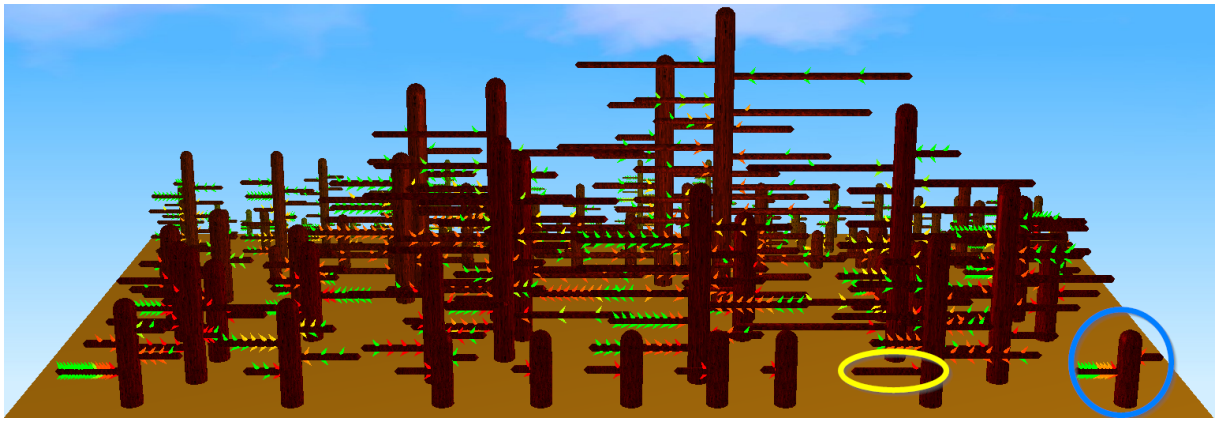


Figure 20 – *The XML-security program represented as a forest of cacti.*

Since the position of the elements in the scene plays a crucial function in capturing the programer's attention, our strategy is to add "weight" to the most suspicious branches, making them closer to the trunk's base. Let us say that the defect is not present in the most suspicious class in Figure 20. The next cactus to be analyzed is at the very left of the most suspicious class in the same row. The first branch to be scrutinized is the one closest to the base as indicated by the yellow circle in the figure.

Unlike other strategies based on tree metaphors (ERRA; SCANNIELLO, 2012), CodeForest branches are always in a parallel position, never facing the developer. By doing so, a bigger number of thorns are shown without the need to rotate or translate the scene.

Nodes, represented by thorns, are alternately positioned on a branch. Starting from the trunk, the first thorn is rendered with the tip pointing upwards. The same exact position holds the second thorn, with its tip pointing downwards. The process continues until all thorns are rendered.

Thorns are positioned taking into account their suspiciousness value determined by a ranking heuristic. The more suspicious a node, the closer its corresponding thorn will be to the trunk. The thorn with the upward tip is more suspicious than another parallel thorn with a downward tip. Interestingly, there is no relationship between the position of a node in the method and its position in the branch. Two "distant" nodes in the code can be close in the branch depending on their suspiciousness values.

The most suspicious class in Figure 20 has two branches, being the most suspicious node the red one closes to the trunk in the lowest branch with the upward tip.

4.2.2 Coloring the elements

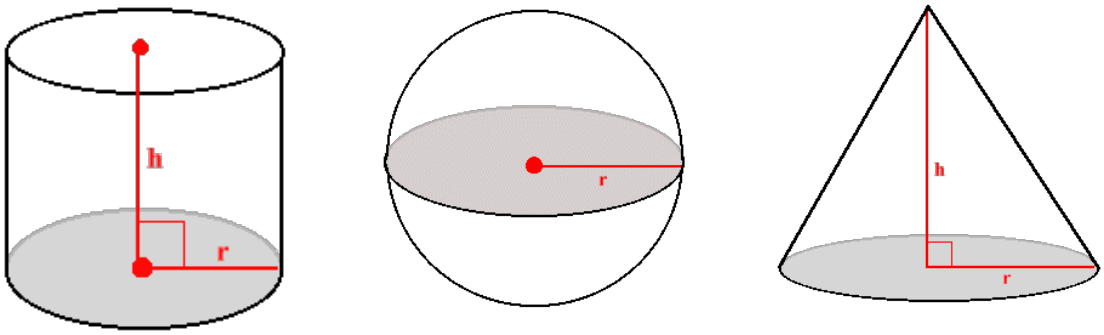
The elements' coloring is determined by their suspiciousness rank. All suspiciousness values are in the interval $[0, 1]$, where 0 is the lowest suspicious value and 1 is the highest one. As a defect is ultimately caused by one or more lines of code (or the absence of it), CodeForest highlights the thorns (the abstraction for lines of code) in the scene by utilizing the same color pattern proposed by Jones et al. (JONES; HARROLD; STASKO, 2002).

The pure green color represents the lowest rank (0). Analogously, the pure red color is associated with the highest ranked elements (1). The intermediate value is yellow and, as the rank moves from 1 to 0.5, the color goes from red to orange and finally pure yellow. When it reaches the lower half, the color varies from pure yellow to pure green.

The coloring of trunks occurs in the same way as thorns. Except that shades used in this operation are darker, both for red and green. Our motivation, as previously stated, is to draw the developer' attention to thorns since cacti and branches are highlighted, especially, by their positioning.

4.2.3 Sizing the elements

The calculation of the height of a trunk takes two class-related properties into consideration: the CH suspiciousness value and the number of methods belonging to it.



(a) Cylinder (b) Sphere (c) Cone

Figure 21 – *CodeForest primitives*

An alternative approach would be to only take the latter into consideration, which could lead to distractions when using it. This would happen in the presence of a class with a large number of methods (for instance, more than thirty) and a low CH suspiciousness value. One example of such situation is the classic Java container class: it has no behavior – just attributes – with a getter and setter method for each attribute. Such a big cactus, even when located in the back of the forest, is able to draw the user's attention to it, yet with minimal chances of containing a defect.

One possible solution would have been to just display cacti with higher chances of containing the defect, leaving harmless classes out of the forest. Although it solves the stated problem, it also creates another one by imposing an obstacle to visualizing the software as a whole. In order to preserve the "whole-system-as-a-cacti-forest" feature offered by CodeForest, we decided that a better strategy would be to change the sizing of elements, based on the suspiciousness value.

Before getting into the details of our solution, we first offer an explanation of how a cactus is build. CodeForest utilizes three building blocks to construct a cactus: cylinders (Figure 21(a)), spheres (Figure 21(b)) and cones (Figure 21(c)). Every cactus in a forest is made from the same unit of construction, a fictitious cylinder with radius R e height H, capable of holding two branches each. The final height of a trunk can be calculated by the equation: $TrunkHeight = (H * (\frac{M}{2})) + H$, where M denotes the number of methods of a class and the extra H refers to the base cylinder, which holds no branches. This process results in a central single-piece cylinder, aligned with the y axis, with a semi-sphere lying at its top (Figure 22).

A branch is made of a single cylinder, aligned with the x axis with a cone as its tip. The radius value used to define the branch is derived from the value of R. The

width of the branch, on the other hand, is randomly picked among several candidates with slightly small variations, where the shortest one is the branch with no free space among thorns and the biggest one is at most two thirds the size of the trunk. Finally, thorns are cones with a fixed radius, also derived from R.

To avoid the problem of huge harmless cacti, we decided to establish the size of the cacti (i.e., the classes) based on their CH suspiciousness value. To accomplish this task, we defined a taxonomy comprised of four categories, presented in Table 10. Then, the minimum values of H and R are determined. Finally, the trunk-building phase was adjusted to give these values a boost, based on the assigned category. As explained earlier, every dimension of every part of the trunk is obtained from the values of H and R. Thus, this strategy resulted in a forest with cacti varying not only in color and position, but also in size and thickness. Additionally, we also defined that each category would also define a maximum size for a branch. Table 11 contains several examples of trunk dimensions (height, radius, and maximum branch size) calculated from the CH suspiciousness value and the number of methods found inside a class.

Table 10 – *Sizing strategy*

Category name	Radius boost	Height boost	Max branch size (cactus height)	Suspiciousness value (CH)
Extra small	$1.0R$	$1.0H$	33%	CH $< .45$
Small	$1.5R$	$1.5H$	33%	$.45 \leq$ CH $< .65$
Normal	$2.0R$	$1.8H$	33%	$.65 \leq$ CH $< .85$
Big	$3.0R$	$3.0H$	66%	$.85 \leq$ CH ≤ 1.00

Table 11 – *Sizing example*

CH	Category	Methods	Height	Boosted height	Max branch size
0	Extra small	10	$6H$ $(\lceil \frac{10}{2} \rceil + 1)$	$6H$ $(6 * 1)$	$2H$ $(6 * \frac{1}{3})$
.56	Small	5	$4H$ $(\lceil \frac{5}{2} \rceil + 1)$	$6H$ $(4 * 1.5)$	$2H$ $(6 * \frac{1}{3})$
.68	Normal	20	$11H$ $(\lceil \frac{20}{2} \rceil + 1)$	$19.8H$ $(11 * 1.8)$	$6.6H$ $(19.8 * \frac{1}{3})$
.89	Big	3	$3H$ $(\lceil \frac{3}{2} \rceil + 1)$	$9H$ $(3 * 3)$	$6H$ $(9 * \frac{2}{3})$

If we take the second line of Table 11, we can see a class which has a CH score of 0.56. According to Table 10, this class will be mapped into a **small** cactus. Because it

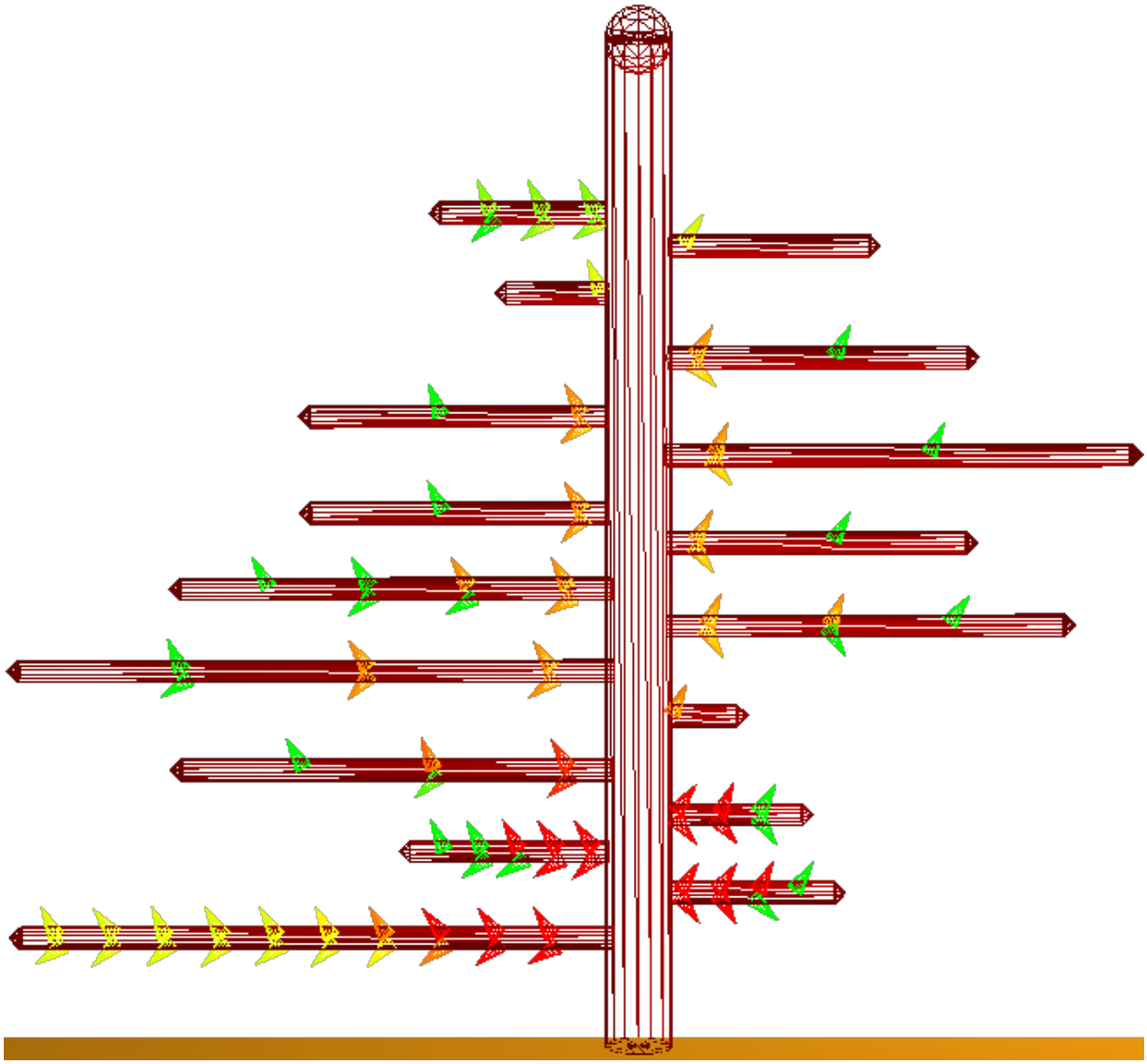


Figure 22 – *Wireframe of a cactus and its elements.*

contains five methods, it will require three cylinders (one for each pair of methods) and one for the base, which gives a calculated height of four H. But, this category receives a small boost in its height (50% in this case), resulting in a final height of 6 H (each cylinder will be 1.50 H high). If we look at Table 10 again, the maximum branch size must be, at most, 33% of the final height, which gives a maximum length of 2 H for each branch.

Figure 22 shows the basic layout of a cactus classified as "Big". After the wireframe rendering stage takes place, the Java3D framework then applies the bark texture on the structure.

4.3 CodeForest prototype

We developed a prototype implementing the CodeForest metaphor to evaluate its usefulness in supporting coverage based debugging. The strategies to position, color, and size the elements of the forest were implemented as well as operations allowing the developer to manipulate these elements to narrow down the fault site. In what follows, we describe the prototype's operations.

4.3.1 Basic resources

Figure 23 shows the XML-security program being visualized using the CodeForest prototype. The screen is divided in three main working areas as indicated by numbers 1, 2 and 3 in the figure. Area 1 occupies the largest space in the screen and is where the forest of cacti is presented. It is also in area 1 where the developer carries out basic manipulation actions available on every 3D prototype, namely, translation, rotation, and zooming.

The source code is available at anytime during the fault localization process. If the developer wants to inspect it, she can click on any element of the scene. If a cactus is clicked the first line of the class is displayed in a text area under the forest (area 2). Similarly, the first line of a method is presented when the associated branch is clicked. Finally, if a thorn is clicked, the interface displays the lines of code of its associated node highlighted accordingly to the node's suspiciousness.

4.3.2 Trimming and filtering operations

In Figure 23, one can observe that the forest of cacti can be quite dense hampering the developer's ability to focus onto highly suspicious thorns, branches, and cacti. Resources to trim and filter elements of the forest were implemented to enhance the developer's ability to localize the fault. These resources are inline with Parnin and Orso's suggestions to leverage coverage based debugging (PARNIN; ORSO, 2011).

The trimming operation allow users to trim cacti, branches and thorns simultaneously or one scene element at a time. The trimming operation is actived when the developer defines both a lower bound and an upper bound of suspiciousness values using a slider available at the right upper corner of the screen (area 3 of Figure 23).

The slider, which we call *trimmer*, controls the suspiciousness of the elements

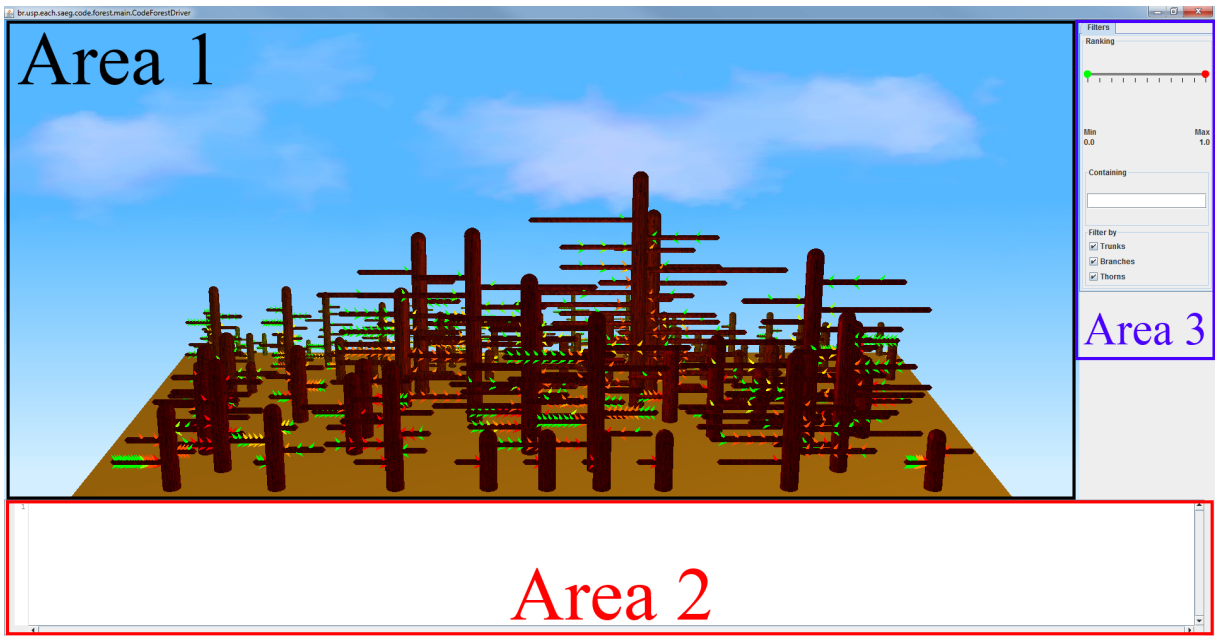


Figure 23 – *The XML-security program visulized with the CodeForest prototype.*

to be presented. All elements whose suspiciousness value lies outside these limits are removed from the forest. The check box in area 3 allows the developer to define which scene elements to be trimmed. By positioning the trimmer at the far right, she will select scene elements with highest suspiciousness value (1.0). As a result, the density of the forest is reduced as shown in Figure 24.

A similar behavior happens when developers utilize the filtering tool. By typing a text in a text box right under the trimmer, the prototype executes a case insensitive search throughout the whole source code. Every element (cactus, branch or node) that contains a match is kept on the scene. To keep the forest metaphor coherent, if a single line of code in a class is a match in a filtering operation, the minimum cactus is displayed, i.e., the cactus, the branch in which that line of code was found, and a single thorn, representing the matching line. Figure 25 exemplifies this situation. The prototype displays the remaining elements after the term "XmlSignatureInput" has been searched.

In Figure 26, a suspect thorn is visualized in detail using the CodeForest prototype. After a zooming operation, its cactus is enlarged and the thorn to be investigated (highlighted by the blue circle in the figure) is clicked. After the click, the lines of code associated with the thorn is presented under the forest (area 2). The color used to paint the code is the same used to paint the selected thorn.

The trimming and filtering operations play an important role on solving one of the most common problems found in 3D environments—the occlusion of elements. With

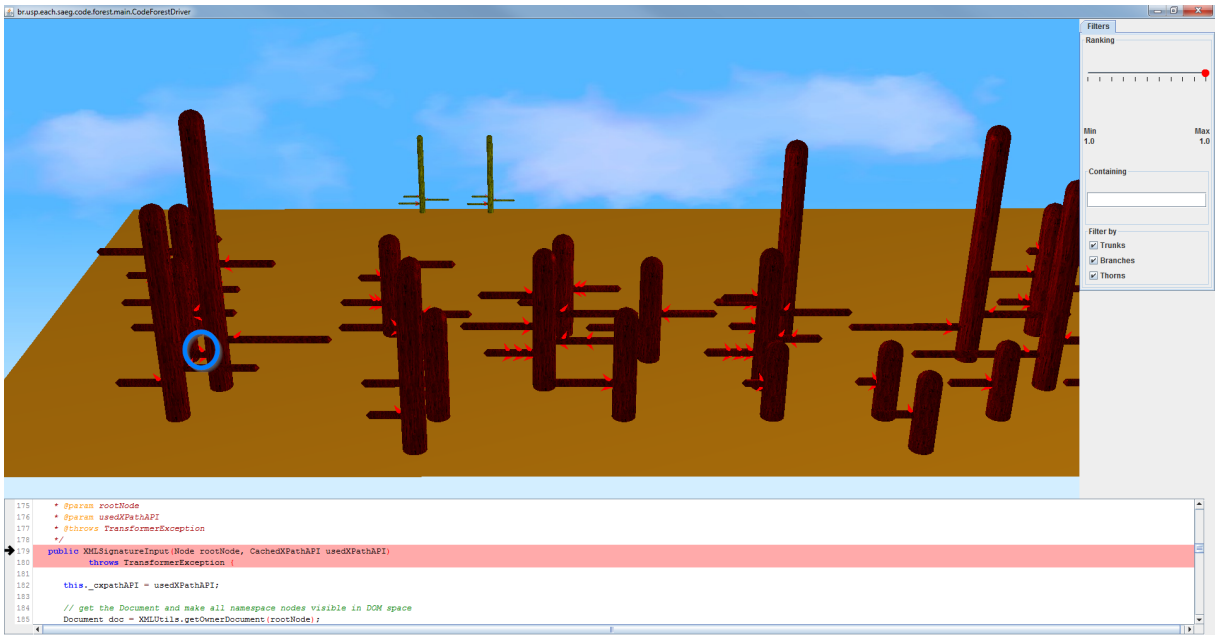


Figure 24 – *View of CodeForest prototype after moving the trimmer to the far right (suspiciousness value equals to 1.0).*

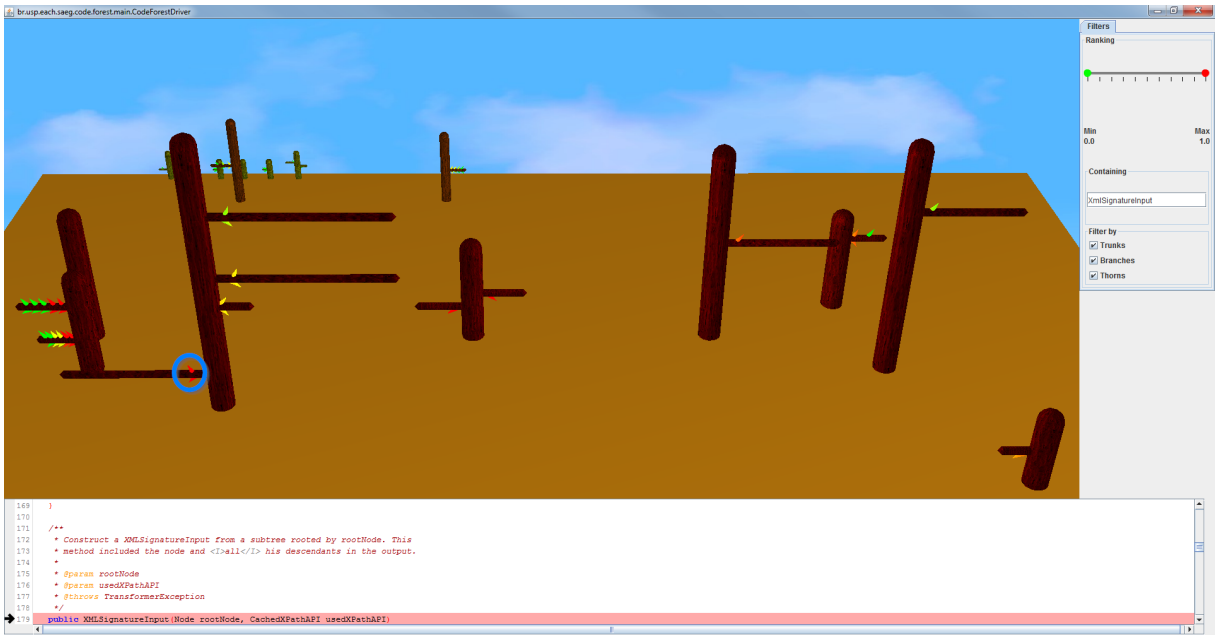


Figure 25 – *View of CodeForest prototype after searching for the term "XmlSignatureInput".*

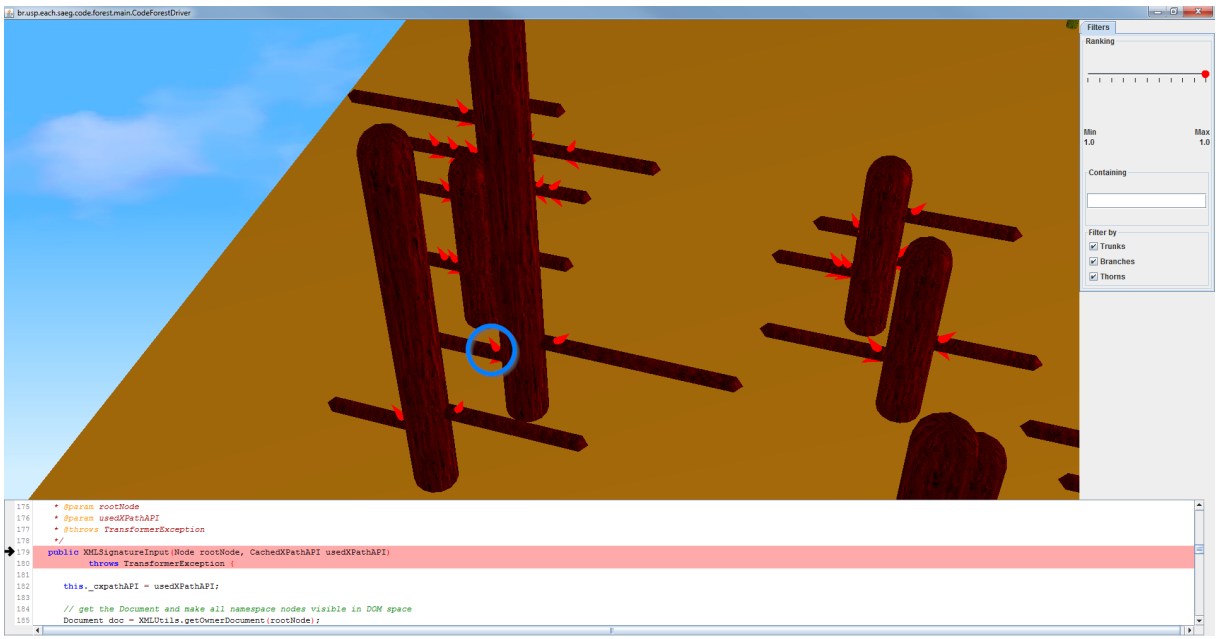


Figure 26 – *Detail of a suspect thorn in CodeForest.*

no elements blocking the object of interest, developers are then free to play with the interface, without making too much effort to select the object of investigation.

4.4 Final Remarks

In this chapter, we presented the CodeForest metaphor. To the best of our knowledge, it is the first 3D based metaphor enabling the visualization of the suspiciousness associated with classes, methods and lines of code (i.e., nodes) in a hierarchical structure: a forest of cacti.

The CodeForest metaphor allows a large scale vision of a faulty system as well as mechanisms to debug it. We have present a prototype in which typical 3D mechanisms (e.g., translation, rotation, and zooming) can be combined with trimming and filtering operations to localize a fault's site.

Unlike other approaches that utilize the same bothanical tree metaphor for software visutalization (KLEIBERG; WETERING; WIJK, 2001; ERRA; SCANNIELLO, 2012), CodeForest supports developers to spot suspicious elements in the software structure and to investigate the piece of code associated with them at the same time. In this sense, it provides a roadmap from suspicious classes, methods and lines of code towards the fault's site.

In the next chapter, we present our approach to deploy the Codeforest metaphor

and its debugging operations in an Interactive Development Environment (IDE).

Chapter 5

CodeForest Plug-in Implementation

This chapter contains relevant details about technical decisions, and the development process adopted, to evolve a proof of concept software into a fully operational plug-in, properly integrated into the Eclipse platform.

5.1 Development process

The development process followed an agile strategy (BECK et al., 2001), which favours principles such as "working software" and "responding to change" over principles inherited from traditional processes like "comprehensive documentation" and "following a plan". Feedback cycles were usually two-week long and most of the documentation is in the form of tickets—both issues and features—submitted via BitBucket[1], a well-known online source code repository.

Development started on September 9th 2013 and ended on April 1st 2014. After 140 commits, a total of 3673 lines of code, 602 methods and 94 files have been written. To preserve the novelty of the idea, a private hosting service was chosen. However, after its publishing, source code will be made fully available to the open source community under the Apache 2.0 license terms[2].

5.2 3D engine technology

The metaphor utilizes the three-dimensional space in order to be able to display large programs (a full explanation is presented in Section 4.1). The choice of a 3D engine was initially bounded by the Java programming language, the underlying programming language used to implement the plug-in. Java3D[3] used to be a very popular choice amidst

[1] `http://bitbucket.org`
[2] `http://www.apache.org/licenses/LICENSE-2.0.html`
[3] `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html`

developers searching for an easy-to-use 3D engine. It is capable of providing a very flexible platform for a broad range of graphics applications (SOWIZRAL; DEERING, 1999). In order to use it, one constructs a virtual universe, and must add to it one or more scene graphs (called BranchGroup by the framework). The virtual universe is a superstructure, comprised of an Universe, a Locale, and (possibly) several scene graphs. A scene graph holds information about which objects to render, the lighs to apply, behaviors to execute, sounds to play, etc (SOWIZRAL; RUSHFORTH; SOWIZRAL, 1997) (Figure 27).

The behavior provides a framework for adding user-defined actions in the scene graph. Java3D scene graph behaviors allows these objects to be automatically rotated, transformed, and scaled using interactive user input. The CodeForest plug-in implements the keyboard behavior, which responds to key-press events that will affect the size, position, and rotation of the scene elements. The keyboard behavior transmits to users the impression that he/she is moving **into** the scene, as opposed to manipulating individual objects **within** a scene, as it would be the case in a mouse-based behavior.

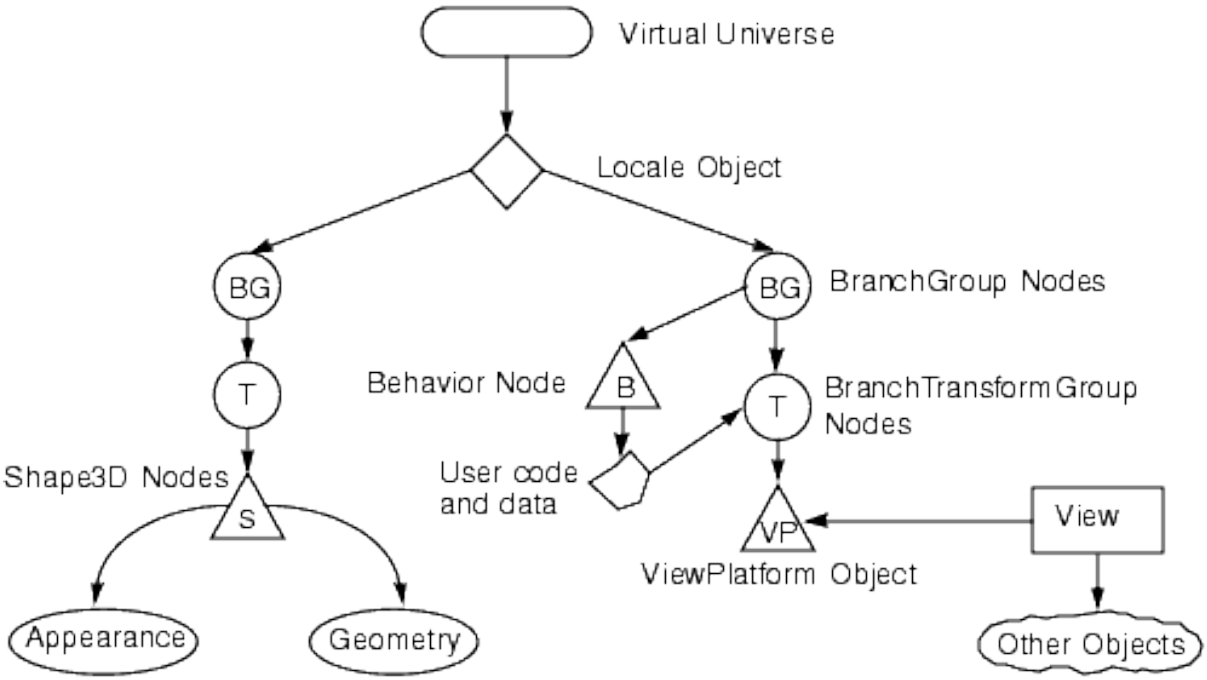


Figure 27 *– A simple scene graph with view control*

Amazon[4], one of the biggest book retailors in the world, accounts 227 programming books containing the term Java3D (AMAZON.COM, 2014a). This is an important factor to be taken into consideration, since 3D Application Programming Interfaces (APIs) are usually non trivial, given the complexity related to the subject.

[4] `http://amazon.com`

The downside of adopting this framework is that Sun Microsystems abandoned it just after its source code was donated, on February 2008, under the terms of the GNU General Public License[5]. However, in 2012, Julien Gouesse[6] ported the last release of Java3D to the Java Binding for the OpenGL API (JOGL) version 2.0[7], a library designed to provide better and faster hardware-supported 3D graphics to applications written in Java. Java3D performance problems remain, and its APIs are outdated (it has not received a single update in six years); but it has several books dedicated to it, not to mention the expressive amount of code examples available on the Internet. These were sufficient reasons to choose it as the plug-in's graphics engine during the development phase.

By the time that this text was being written, a new version of Java, the eighth one, has been released by Oracle Corporation [8]. The relevant fact is that the new version of JavaFX that is shipped with the latest JDK now includes 3D Graphics features such as shapes, camera, lights, subscene, material, picking, and antialiasing. This release makes JavaFX a strong candidate to substitute the old Java3D technology, which was used to develop the prototype and the plug-in. Though, more time is necessary to confirm if this technology will become the *de facto* standard of 3D development in Java.

5.3 Hosting environment

The task of choosing an Integrated Development Environment (IDE) to host the CodeForest plug-in started with three candidates, currently the most popular IDEs amongst developers (FURMANKIEWICZ, 2008): Eclipse (FOUNDATION, 2014a), NetBeans (CORPORATION, 2014), and IntelliJ IDEA (JETBRAINS, 2014); all of them available on the Internet.

The three following questions were prepared in advance to aid our search to find the most adequate IDE to integrate with. They embody the aspects that are considered most relevant to the decision process.

1. Which IDE has the highest market share?

2. Are all IDEs free (as in "free speech")?

3. Which one of them has the largest body of documentation available?

[5] http://www.gnu.org/licenses/gpl-2.0.html
[6] http://gouessej.wordpress.com/2012/08/01/java-3d-est-de-retour-java-3d-is-back
[7] http://jogamp.org/jogl/www
[8] http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Although there is not a large body of research to corroborate it, Eclipse seems to be the winner of the Java IDE Wars (GEER, 2005). A research conducted in 2012 found that, of 1800 participants that took part in an online survey, 70% pointed Eclipse as their primary development environment, followed by 28% of IntelliJ IDEA users and 17% of NetBeans users (KABANOV, 2012).

When analyzed under the perspective of price, both Eclipse and NetBeans stands apart from IntelliJ IDEA. Jetbrains offer two editions of its IDE: Community and Ultimate. The Community Edition works only with Java Standard Edition and it has no cost. The Ultimate Edition works with both Java Standard and Enterprise Edition and it costs US$ 199 for individuals and US$ 499 for companies and organizations[9]. Both Eclipse and NetBeans are open source products, freely offered by the Eclipse Foundation and Oracle Corporation, with no restrictions of use.

From the documentation perspective, Amazon has for sale 286 Eclipse related titles (AMAZON.COM, 2014b). NetBeans can be found on 136 books (AMAZON.COM, 2014d) while IntelliJ Idea is the subject of a total of 23 books (AMAZON.COM, 2014c). This method to find out which one has the biggest body of documentation available falls short of scientific rigor but it can provide us with a strong indicator of which IDE is most documented.

When we take into consideration the questions initially defined and their answers, the Eclipse IDE stands out as the most adequate platform to host the CodeForest plug-in. It is the *de facto* standard amongst developers, it is free and freely distributed, and—according to our criterion—it has the largest body of documentation available. These points are responsible for making Eclipse our IDE of choice. Sections 5.4 and 5.5 provides a brief explanation about Eclipse, its history, and the underlying technology about its runtime environment, OSGi.

5.4 Eclipse

Eclipse began in 1999 as an IBM Canada project, whose goal was to unify all of its development environments on a single codebase. Many of IBM's development tools were written in Smalltalk[10], which was rapidly losing market share to languages like Java. IBM's VisualAge for Java IDE, named "Best Product of the Year" by InfoWorld in

[9] Obtained from `http://www.jetbrains.com/idea/buy/index.jsp` on September 9th of 2014.

[10] Smalltalk is an object-oriented, dynamically typed, reflective programming language designed and created in the 1970s at the Learning Research Group (LRG) of Xerox PARC (MYERS, 1998).

2001[11], was written in Smalltalk. The problem is that, except for IBM and a few other vendors, the language did not have significant adoption in the industry. Thus, IBM tasked its Object Technology International (OTI) group with creating a highly extensible IDE based in Java. The Eclipse Platform goal was to eclipse its competitor, Microsoft's Visual Studio (CLAYBERG; RUBEL, 2008).

At first sight, Eclipse might look like a giant piece of software—the latest release, codename Luna—is estimated to have 61 million lines of code (TAFT, 2014). Contrariwise, it is structured as a small kernel containing a plug-in loader, surrounded by hundreds of plug-ins.

The fundamental concept about the Eclipse platform is that everything is a plug-in, where each subsystem that is part of the Eclipse Platform is itself structured as a set of plug-ins that implement some function. Eclipse Software Development Kit (Figure 28) includes the basic platform plus two major tools designed for plug-in development. One is the Java Development Tools (JDT) which implement a full featured Java development environment. The other is the Plug-in Developer Environment (PDE), responsible for adding specialized tools to leverage the development of plug-ins and extensions (BROWN; WILSON, 2011; FOUNDATION, 2014b; D'ANJOU, 2005).

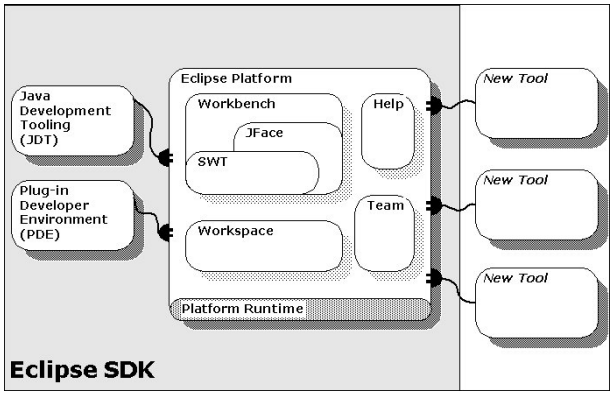


Figure 28 *– Eclipse SDK architecture*

During the development of its third release, the Eclipse Foundation created a sub-project called Equinox, aiming at replacing Eclipse's homegrown runtime platform (called component model) with one that already existed, as well as providing support for dynamic plug-ins. The team decided to implement the OSGi (Section 5.5) specification as the underlying system in charge of managing plug-ins. The rationale was to appeal to a broader community by standardizing on a component model that had wider adoption outside the Eclipse ecosystem (BROWN; WILSON, 2011). Migrating the original component

[11] `http://www-03.ibm.com/press/us/en/pressrelease/1799.wss`

model to OSGi was the last major change made on Eclipse's infrasctructure (BLEWITT, 2013).

5.5 OSGi

Formerly known as the Open Services Gateway initiative (OSGi), the OSGi Alliance was formed in March 1999 by a consortium of leading technology companies with the mission to define a universal integration platform for the interoperability of applications and services. OSGi provides a development platform based on modular decoupled components and a pluggable dynamic service model for the Java platform. A development platform, in the context of this research, is a set of software libraries and tools that aid in the development of software components, and the corresponding runtime environment that can host these developed components.

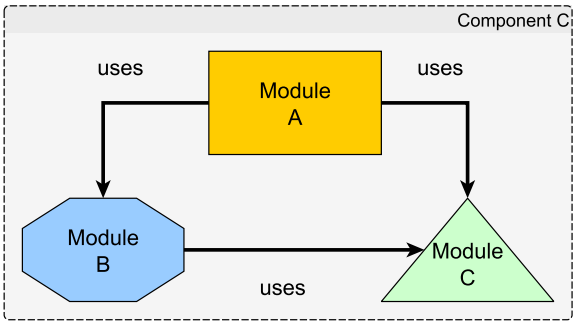


Figure 29 – *A component decomposed into modules*

OSGi was conceived to target the lack of advanced modularization support in standalone Java VM environments. Modularity here refers to the logical decomposition of a large system into smaller collaborating pieces (Figure 29). Java provides some aspects of modularity in the form of object orientation, though Sun Microsystems did not make any plans to add support for programming (HALL; PAULS; MCCULLOCH, 2011; ALVES, 2011). There are plans to add a modular system to the Java VM and, according to the most recent schedule, it will be available in Java 9[12]. Meanwhile, the resources available to developers are the language's standard mechanisms: visibility modifiers—public, protected, private, and package private—and packages, usually employed for purposes of partitioning code. Both of them solve low-level object-oriented encapsulation issues (GOSLING et al., 2005). However, neither of them are suited for logical system partitioning.

[12] `http://openjdk.java.net/projects/jigsaw/`

Another modularity issue present in the Java platform is the classpath; it generally inhibits good modularity practices. The classpath is a location, a place where the compiler will look for classes that the running application demands to be loaded during runtime. In general, an application depends on various versions of libraries and components. The classpath does not offer a way to declare which code version a given application relies on; it returns the first version found. As a result, large applications that depend on different source code versions of the same library will probably fail at some point of their lifetime. These kind of failures happen during runtime, from causes such as NoSuchMethodError (when a wrong version of the class is loaded) or ClassNotFoundException (when a wrong version of the library is loaded).

The OSGi framework is divided into three layers: module, lifecycle and service. The **module layer** defines the OSGi module concept (called *bundle*). A bundle is a JAR file, with its own class loader, that contains extra metadata which allow the developer to explicitly select packages that are going to be visible to other packages. This mechanism extends the access modifiers (public, private, and protected) provided by the language. The **lifecycle layer** administers the application, supplying controls to safely add and remove bundles without the need to restart the application. Additionally, this layer lets components to be "lazy loaded"; in other words, a component is loaded by the application only when required (BROWN; WILSON, 2011). Finally, the **services layer** provides a way to publish services into a registry, while clients search the registry to find available services to use. Services can appear and disappear at any time (HALL; PAULS; MCCULLOCH, 2011; ALVES, 2011).

The flexibility, advanced controls of managed resources, and lifecycle control offered by the OSGi framework make it the perfect foundation for the Eclipse environment, whose main purpose is to be highly extensible, stable, and consume minimum computational resources.

5.6 Dependencies

To complete this research in a reasonable time, certain tradeoffs had to be made, specially when it comes to transparently integrate the plug-in with other tools developed by the members of the Software Analysis & Experimentation Group (SAEG).

One of such a tools is the Integration Coverage-based Debugging (ICD) tool (SOUZA; CHAIM, 2013; SOUZA, 2012). ICD tool receives as input the MCP and block

coverages (Section 2.2) of the test suite, the chosen heuristic and the outcome of the tests (success or fail). It outputs the MCP and block lists and the roadmap. The input is generated by Instrumentation Strategies Simulator (InSS) (SOUZA, 2012), which instruments the source code, executes unit tests, and gathers the coverage data and the outcome of every test run. Figure 30 describes how the InSS and ICD tools work together to produce the guidelines for fault localization based on integration coverage.

The result of this process is outputted as an XML file, containing a set of tuples. Each element of this set is a pair of (1) an element of the source code (node, method, or class) that was exercised by unit tests and (2) a suspiciousness score, assigned based on the results of applied heuristics.

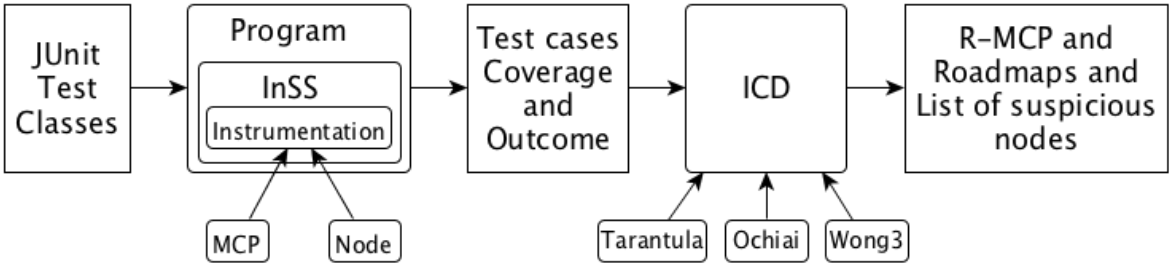


Figure 30 – *ICD implementation*

The current development state of the plug-in requires the execution in advance of the ICD/InSS workflow and that the generated file be copied into the project's root folder as "codeforest.xml". It is our intent to make these steps transparent to the user in the future, absorbing them as a part of the plug-in's functionality. It is expected that this feature would shorten feedback cycles of the bug-fixing activity (Figure 31).

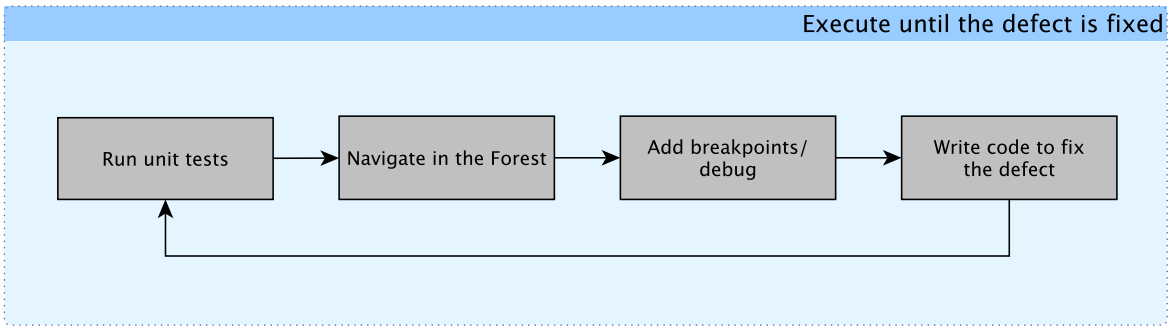


Figure 31 – *Bug-fixing feedback cycle*

5.7 Plug-in operation

As previously explained, the plug-in is activated by the presence of a file generated by the InSS tool (SOUZA, 2012), named "codeforest.xml" and located at the project's root directory (Figure 32). When this file is present, the developer that is operating the Eclipse IDE can right-click the project under analysis.

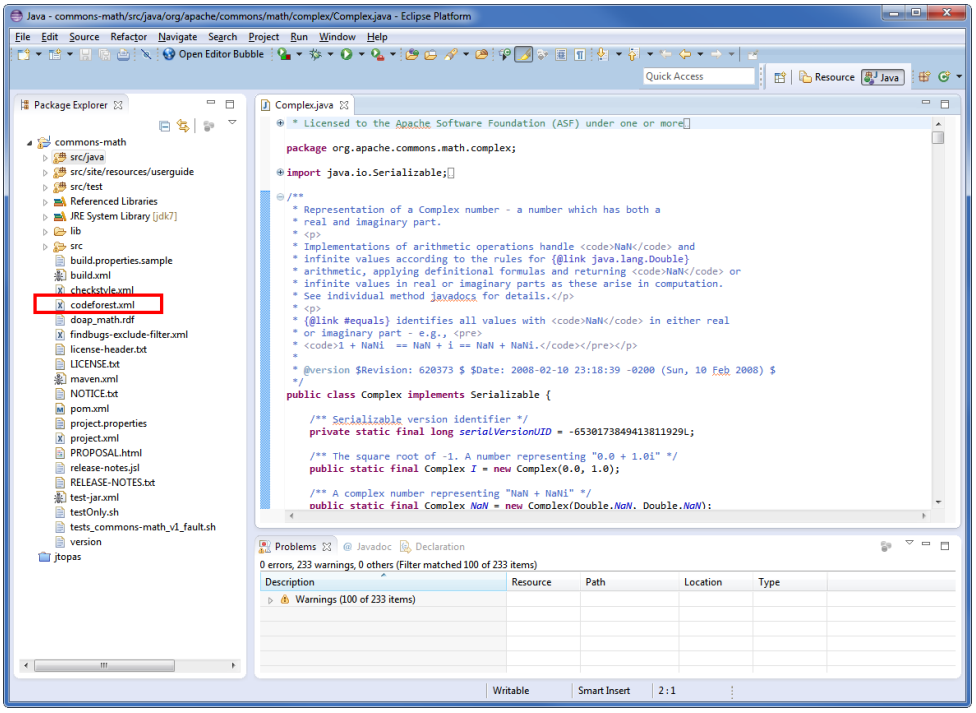


Figure 32 *– A codeforest.xml file of Apache Commons Math project*

When this action is fired, a popup menu is displayed; amongst its several options, the CodeForest submenu is shown. At this point, the developer could ignore the execution and continue his/her task at hand or select the "Perform analysis" option (Figure 33).

A single left-click on the "Perform analysis" command runs a series of tasks, whose output is that every Java file opened for edition is decorated, i.e., its statements are painted according to suspiciousness values read from the XML file (Figure 34). Internally, the plug-in parses the Java source code, identifies statements and searches for its related data obtained from the input file ("codeforest.xml"). This architecture enables the plug-in to display different data using the same infrastructure; it only requires new adapters to read the data source and adapt it to the internal domain objects.

From this point onwards, the developer can keep the highlight hints to aid debugging the system and/or navigate the codebase using the CodeForest features (Figure 35).

Choosing to display the CodeForest instructs the plug-in to open a smaller version

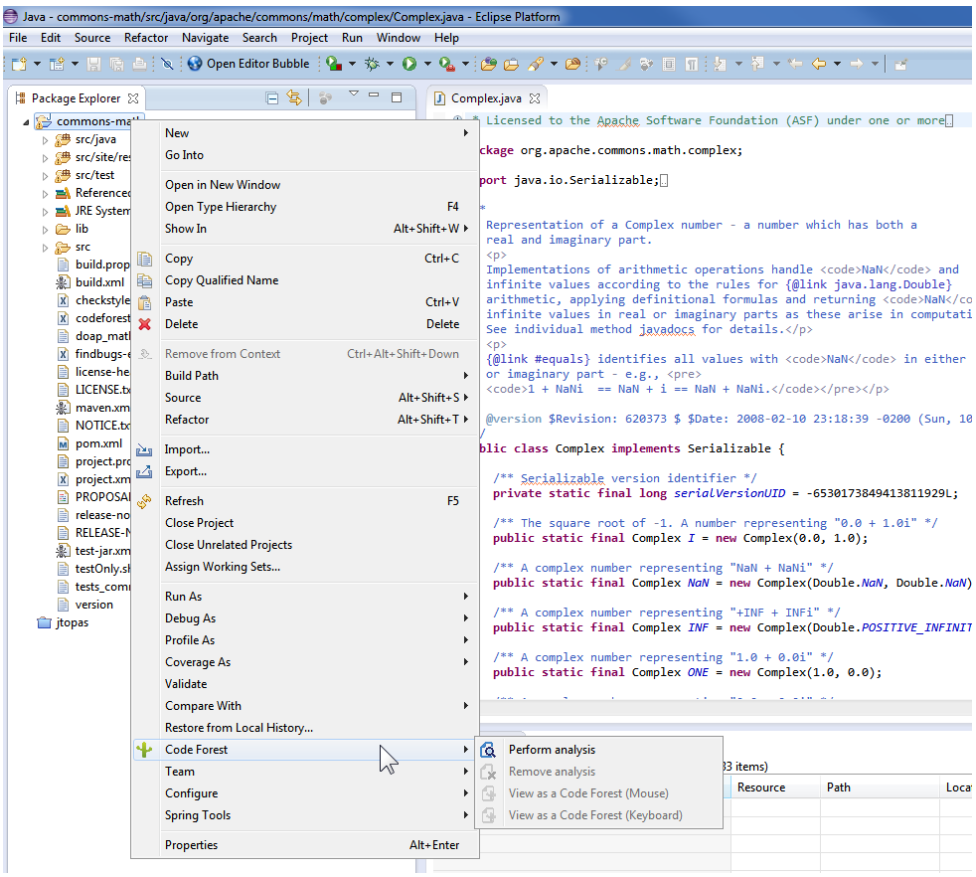


Figure 33 – *The CodeForest menu expanded*

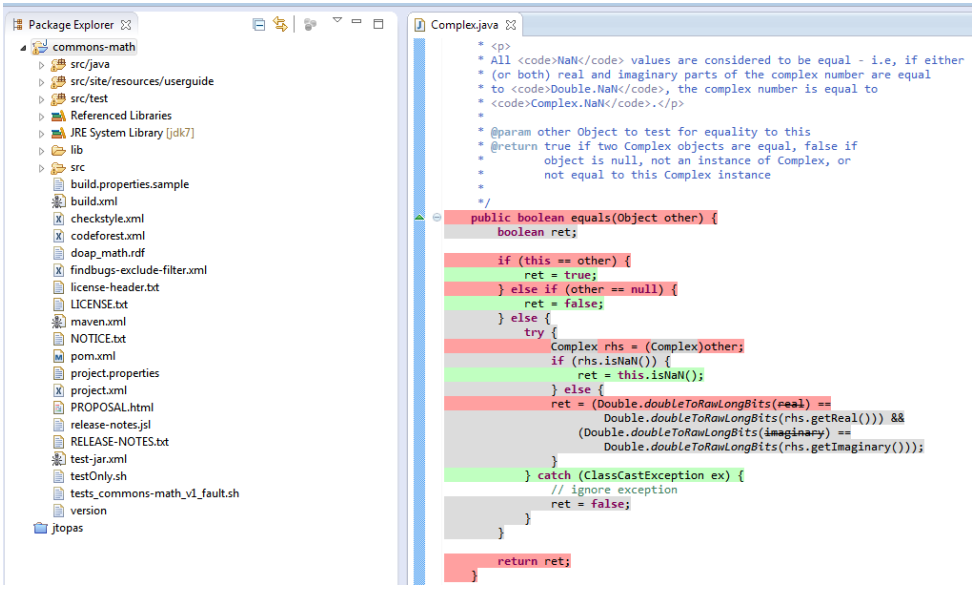


Figure 34 – *A method highlighted by the plug-in*

of it, still embedded inside the Eclipse IDE (Figure 36). From our observations during experiments with the tool (Chapter 6), the recommended way to explore the CodeForest is to detach it and drag the window into a separate screen. Setting up the debugging environment this way allows the developer to freely explore the forest, while keeping the

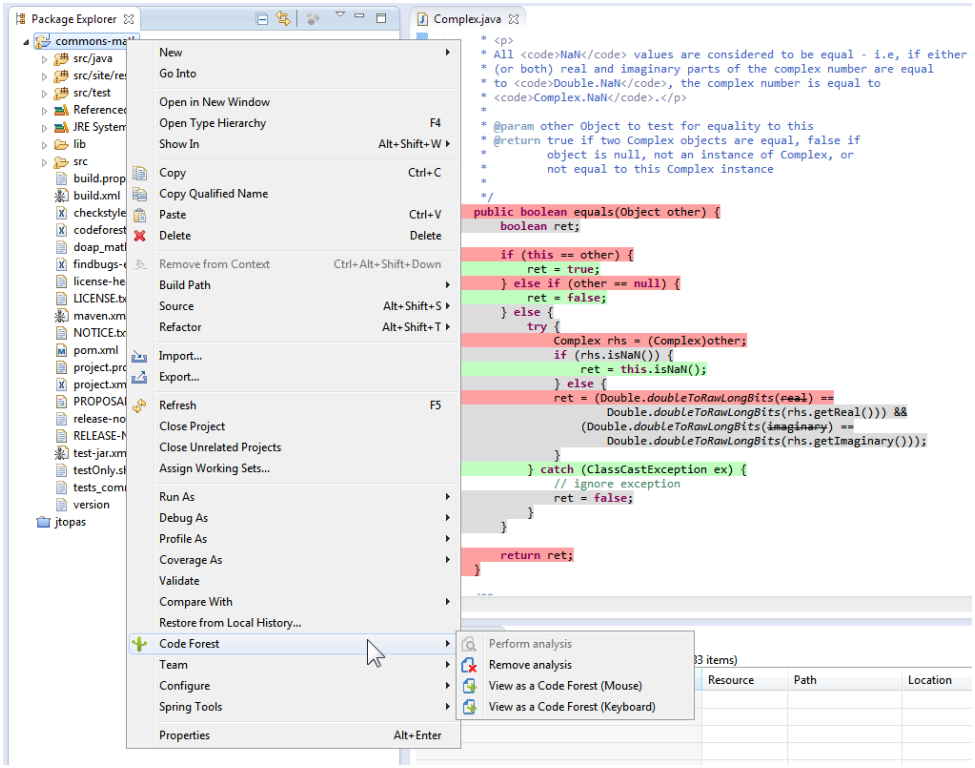


Figure 35 – *Menu options after performing project analysis*

source code visible all time, and vice-versa.

The tools present in the prototype (trimmer and filter) remained a part of the plug-in, and their functions were preserved. The trimmer (Figure 37–Area 2) excludes elements from the scene whose ranking is out of a delimited range: by moving two sliders, the developer defines the inferior and superior limits. The filter (Figure 37–Area 3) is responsible for searching the codebase, looking for elements that contain the inputted term. It was implemented as a simple, case-insensitive, textual search. Every element that does not contain the queried term is excluded from the scene. If there is one line of code in a whole class that contains that term, then the forest will be reduced to a single cactus, with one single-thorned branch. Both tools can work together or independently.

The third tool present in the plug-in is the roadmap (Figure 37–Area 4). It contains a sequence of steps (methods) that the developer is advised to follow in order to find the defect. Items are painted with the color related to its ranking (red, orange, yellow, and green). The roadmap is comprised of a limited list, ordered in decrescent order of suspiciousness, and is not a tool per se, but a kind of smart shortcut to the other tools. When one of the items of the list is picked, both trimmer and filter are parameterized with data from that element. The term to be searched is the name of the method, excluded its signature—the method “createComplex(double,double)” sets up

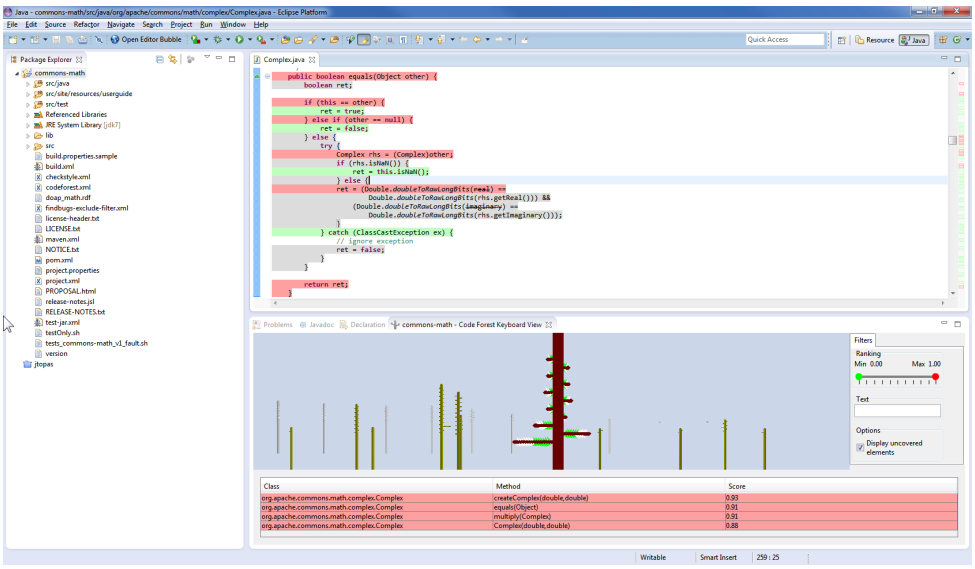


Figure 36 – *CodeForest embedded inside Eclipse IDE*

the query "createComplex". The trimmer's range is delimited inferiorly by the method's suspiciousness score and superiorly by the same score added of a 3% delta (SOUZA, 2012).

Finally, there is the CodeForest (Figure 37–Area 1), the virtual environment that lets developers navigate through the codebase, represented as a forest of cacti. The rationale behind the construction of the forest is explained in details in Chapter 4. The developer interacts with elements using keyboard and mouse.

The keyboard is used to move the developer through the forest. Arrow keys move the observer forward (up), backward (down), left (left), and right (right). In this kind of basic movement, the observer always face forward. Modifier keys change the nature of the movement. Alt key combined with one of the arrows makes the observer move up or down, emulating the climbing or descending of a ladder. Control (Windows and Linux) or Command (MacOS) key emulates the "neck" movement, making the observer thinks that his or her head is moving up, down, turning left or right.

The developer uses the mouse in a "point and click" fashion. When in need of opening the source code file associated with any desired element, he/she clicks it with the mouse's left button. This action fires a task that opens a Java Editor window containing the Java file related to that element, and positions the cursor at the appropriate location—a click on a cactus will put the cursor at the line where the class is declared; on a branch positions the cursor at the method's declaration; a thorn will open at the corresponding line of code.

At any time, the developer can access the CodeForest menu by right-clicking

the project and selecting the “Remove analysis” option. This will close the CodeForest window and remove the highlight of every file within the current project.

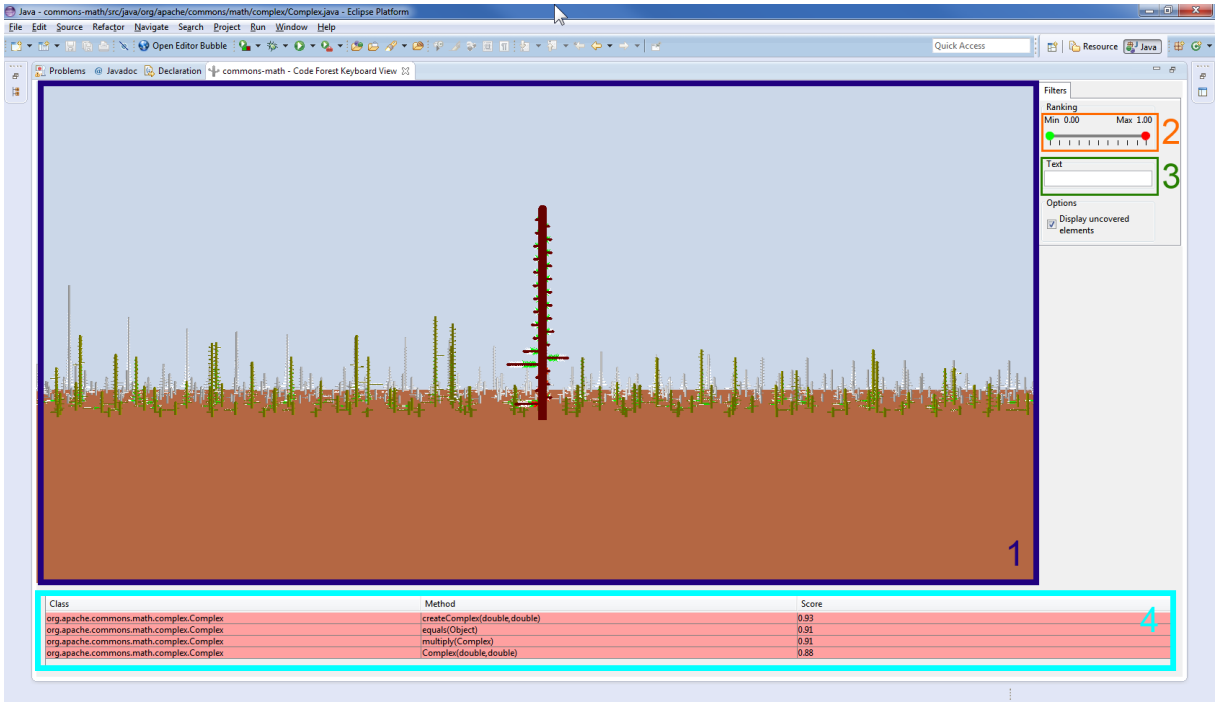


Figure 37 – *Maximized view of CodeForest and its elements*

5.8 Final remarks

This chapter presented the architectural and technical forces that guided the development of the CodeForest as a plug-in embedded on an Integrated Development Environment (IDE). It also explained the external dependencies and how they integrate with the plug-in. A brief explanation of the tools that are part of the plug-in and how to utilize them was also given.

The following chapter describes the experiment that has been set up to evaluate the plug-in. It points the opinions, insights and any other information gathered that could guide future enhancements and adjustments in the tool.

Chapter 6

Experimentation with CodeForest

This chapter describes the set up, implementation, and execution of an exploratory experiment. Our goal was to observe how the CodeForest plug-in would be used by developers commissioned with the task of investigating a piece of software containing a defect that made unit tests fail. To accomplish that goal, four individuals with different proficiency levels in Java and unit tests were selected. The volunteers that took part in the experiment received two distinct investigative tasks: the Thoughtworks XStream (WALNES et al., 2005) (referred to XStream from now on) object serialization framework and the Apache Commons-Math (COMMONS, 2013) (referred to Commons Math from now on) library. The assigned order of tasks varied from volunteer to volunteer, though the defects were always the same.

The following sections details the whole process employed to realize the experiments and their outcomes.

6.1 Experimental design

In what follows, we discuss the research questions prepared in advance, the desired participants' profile, the objects, and the procedure employed to carry out the experiment.

6.1.1 Research questions

This section starts with the the questions that we wanted to answer through the experiments. They were the following:

- Is the metaphor adequate to the problem of locating defects?
- Are the tools embedded within the plug-in useful?
- Are there tools more useful than others? If so, which ones?
- What is the dynamics between the developer and the tool when he/she is trying to locate defects?

6.1.2 Participants

In order to answer these questions we set up, implemented, and executed an exploratory experiment. It began with the definition of the target audience, the kind of people that would use the CodeForest plug-in. As previously explained, the CodeForest plug-in is hosted by the Eclipse Platform. This way, participants would have to, at least, be willing to investigate a codebase written in Java, using the Eclipse IDE. Also, since this research proposes a new way to display fault localization data, it is desirable that volunteers had previous knowledge in debugging techniques. Four individuals from the Software Analysis & Experimentation Group (SAEG) volunteered to be part of the experiment.

6.1.3 Objects

The preparation of the experiment demanded the identification of programs to be debugged. The software that was going to be presented to developers, the debuggee, must have a) JUnit (GAMMA; BECK, 2006) tests—that is the input of the INSS (SOUZA, 2012) library, which calculates the suspiciousness value of software elements (an in-depth explanation on this subject is available in Section 2.2); and b) tests that test small units of code, so they can enable developers to narrow their search to suspicious lines of code, instead of coarser structures, such as methods or classes.

Besides those restrictions, questions of technical nature arise specifically related to the troubled code: a) which defect would be embedded on the software? b) should it be artificial (introduced via random modifications in the source code) or real (from a ticket in a bug tracking system)? c) something simple as a misplaced operator or difficult as a missing line of code?

To gain as much insight as possible from the experiment, each developer would have to spot the defective line in two different scenarios: a misplaced operator planted in XStream, inspired by (CAMPOS et al., 2012); and a real missing "if"clause, mined from the Subversion history of the Commons Math project. These projects were selected based on their quantity of JUnit tests, coverage data, and their complementary nature. XStream builds a long chain of invoked methods to (de)serialize objects. Commons Math, on the other hand, is comprised of several independent methods, and most of them are implementations of mathematical functions.

Commons Math is made of 26,202 lines of code, of which 92% covered by 1,172

unitary tests (124,523 instructions of 135,331) according to Eclemma—a Java coverage tool for Eclipse (HOFFMANN, 2009). The same tool reported XStream as less covered: 85% by 1,492 test cases (72,322 instructions of 84,589) in a total of 25,385 lines of code.

6.1.4 Procedure

This subsection describes the procedures utilized to collect the data analyzed in this experiment.

6.1.4.1 Training

Firstly, we carried out a training session that every participant attended before engaging into any debugging session. The slides used in this activity can be found on Appendix B. It focused on the transmission of core concepts involving this research, such as code coverage, visual debugging, the experiment that was going to happen, and the plug-in itself. The explanation took no more than thirty minutes and was available for consultation (as an electronic document) during the whole experiment.

6.1.4.2 Environment

The environment where the experiment was conducted is described below.

- The participant remains alone in the room, and is free to abandon the experiment at any time.
- The experiment is divided into two parts. In each one of them, the participant have to point the line of code (or the absence of it) that caused the defect to occur.
- The defect does not need to be corrected.
- The first task of the experiment is the XStream (or Commons Math).
- Once started, the participant has sixty minutes to find the defect.
- When the participant reaches a conclusion, the second part begins. It works as the first part, except that this time, the task has a maximum duration of thirty minutes.
- By the end of the experiment, the participant is asked to fill a questionaire (Appendix C) and report any suggestions or corrections.

The first task had the double of the duration of the second duration to allow participants to move up the learning curve. Our assumption was that these extra thirty minutes would be enough to go from "complete uncertainty" about the experiment to a level of certain confidence in it.

The computer used to perform the tasks was a MacBook Pro with a 13-inch display equipped with a secondary 20-inch display. Volunteers were advised to run Eclipse on the notebook screen and the CodeForest (if opened) on the 20-inch screen.

All participants selected to take part in the experiment were asked to read and sign the document "Informed Consent Form" in advance, which gave us permission to collect data and use it on scientific publications, as long as participants remain anonymous. An exact copy of the document is available on Appendix A.

6.1.4.3 Data collection

Part of the answers to the questions formulated at the beginning of this chapter are the actions performed by users when interacting with the plug in. In order to collect this data, we opted to perform an instrumentation that would log significant actions taken by developers. This modification should suffice to understand the behavior of the person who is interacting with the plug-in. These were the events logged during usage:

- Keystrokes pressed during the navigation in the forest.
- Mouse interactions with the elements of the forest.
- Start and end of debugging sessions.
- Addition or removal of breakpoints.
- Clicks on items of the roadmap.
- Changes applied in filters (text and/or score).

The questionnaire (Appendix C) participants were asked to fill contains 25 questions, 20 of which have their options modeled as a five-degree Likert scale (ALBAUM, 1997), varying from "Strongly disagree" to "Strongly agree". They inquire about aesthetics, usage and general perception of the tool. The remaining 5 questions are related to the experience of the participant, with options such as "none", "0 to 2 years", and so on. The questions that comprised the questionnaire were non-standard and, in the future,

should be replaced by standardized ones. This is necessary to avoid inducing positive or negative bias in the answers.

6.2 Pilot experiment

Our pilot experiment served as a "validation of the experiment design". The participant, a Master's student, had approximately one year of professional experience in development of enterprise Java programs, investigation and correction of defects in code authored by others.

The plug-in operated during the pilot experiment had two implementations of the 3D forest environment. One, operated with a mouse, was basically a port of the original forest environment found in the CodeForest prototype (Chapter 4). Other, operated with a keyboard and mouse, was developed after suggestions made by an expert in the field of 3D Human-Computer Interaction (HCI) that evaluated the prototype.

6.2.1 XStream

The participant related an attempt to start the defect investigation with the CodeForest Mouse View but quit it shortly after the beginning, claiming that it was too difficult to operate (the time spent on this attempt was ignored during result compilation phase). Next, he opened the CodeForest Keyboard View and continued the task using that environment. He failed to find the defect on the first thirty minutes and, when asked, agreed to extend the task.

It took a total of thirty four minutes to complete the first task. He spent nearly one and a half minute exploring the forest; selected elements of the roadmap seventeen times, and clicked eight times on thorns. He also spent approximately twenty minutes debugging the code. In the end, the volunteer succeeded to point the exact line that originated the defect.

6.2.2 Commons Math

After successfully completing the first task, the volunteer received the next challenge. The defect in question was caused by the absence of an "if" statement in the Commons Math codebase. This framework differs from the XStream in the sense that it is made of several independent units, with little or no relation amongst themselves.

This structure creates a shorter chain of methods to be analyzed. It did not make the task easier, though. The number of classes and methods found in the framework confuses developers that are not used to its source code.

He spent a total of thirteen minutes to complete this task. Nearly two minutes were spent interacting with the forest. He selected elements of the roadmap sixteen times, and clicked nineteen times on thorns. This time, the volunteer spent two and a half minutes debugging. In the end, the volunteer succeeded to point the exact line that originated the defect.

6.2.3 Summary

The participant was able to point the line containing the defect on both frameworks investigated. Table 12 summarizes data collected from instrumentation logs of the two debugging sessions. The header indicates the library under investigation, namely XStream and Commons Math. Underlined items are the events that were logged during the experiment. Below each item there is one or more numbers, and on the left of each number, a short description of the aspect being summarized. Finally, the footer contains the answer to the question "Was the defect found?". Following, there is a brief explanation of each event and their respective aspects that were considered in this research.

- **CodeForest** refers to the events that happened inside the three-dimensional virtual forest. "Time exploring" is the total amount, in minutes, spent navigating through the forest; "Movements" is the number of keys pressed to move the virtual camera, like moving it forwards or upwards.

- **Debugging** refers to the debugging activity, i.e., the execution of lines of code, one statement at a time. "Time spent" is the sum of all time spent by the volunteer on this activity. "Events" is how many times the user started and ended a debugging session.

- **Breakpoints** refers to the act of adding or removing breakpoints into the code. A breakpoint is a point in space (line of code) where the debugger pauses the execution, so the user can inspect the runtime state of the program.

- **Roadmap** refers to how many times ("Events") the user clicked on elements of the roadmap, presented as a table on the bottom of the screen. This number is absolute; two clicks on the same item counts as two events.

- **Trimmer** refers to the filtering tools that allows users to filter elements of the forest based on suspiciousness and text. This is intended to enhance focus by reducing the number of elements (cacti, branches, and thorns) being displayed. "Adjustments (text and score)" is the number of times that the user made adjustments on the parameters of the filter: minimum and maximum score and text. This number is not related to the "Roadmap" functionality, which also triggers changes on filters.

- **Elements** refers to the elements of the forest that were clicked by the user. When the user interacts with a cactus, the IDE opens a window with the cursor positioned at the first line of the file represented by the cactus. A similar behavior happens when branches or thorns are clicked.

- **Reset** refers to the number of times ("Events") that the user pressed the key to reset the forest. This functionality acts as a "panic button". It repositions the camera, and redefines the trimmer (minimum, and maximum score, and text filters) to their original values, respectively "0.0", "1.0" and empty text.

- **Task** refers to the total time, in minutes, necessary to complete the proposed task.

Table 16, described in Appendix D, contains the answers provided after finishing the tasks.

The feedback provided by the volunteer motivated changes in some characteristics of the plug-in that were implemented based on unconfirmed assumptions. One of the feedbacks questioned the divergence between the behavior of up and down arrows (to move forwards and backwards) and left and right arrows (to move the neck). Based on that, we changed the behavior of the left and right keys to move the observer laterally when pressed.

Another characteristic that was modified motivated by the feedback was the approach utilized by the plug-in to generate the CodeForest. The participant utilized a version that rendered the whole codebase as cacti, including statements, methods and classes that were not exercised by unit tests. In his opinion, this feature cluttered the visualization, undermined the performance (due to the quantity of objects on the scene), and diverted his attention to elements that were not relevant in the context of the defect being investigated.

Other suggestions need further confirmation due to the fact that they affect the basis of the metaphor, such as changing the positioning strategy of the branches. In the

Table 12 *– Pilot experiment summary*

	XStream	**Commons Math**
	CodeForest	
Time exploring	01:14	02:43
Movements	637	355
	Debugging	
Time spent	20:31	02:34
Events	14	4
	Breakpoints	
Added	10	1
Removed	7	0
	Roadmap	
Events	17	16
	Trimmer	
Adjustments (text and score)	45	2
	Elements	
Cacti	12	4
Branches	1	1
Thorns	8	19
	Reset	
Events	5	0
	Task	
Duration	34:00	13:00
Defect found?	Yes	Yes

participant's opinion, instead of being arranged by their suspiciousness, branches ought to be arranged by their position in the source code.

6.3 Experiments

After performing the adjustments that emerged from the pilot experiment and improving the user interface, we started the preparations for another round of experiments. These experiments happened on the same day, one after another, and followed the same dynamics applied on the pilot experiment.

Every participant did the experiment on the same place, using the same machine (a 13-inch MacBook Pro with a 20-inch extra monitor), had the same training and the same challenges: to find the defect on the XStream and the Commons Math frameworks. Two participants started with the XStream and finished with the Commons Math framework. The remaining one did the opposite; he started with the Commons Math and ended

with the XStream framework.

6.3.1 Experiment # 1

Participant #1, a Master's student with a bachelor's degree in Applied Mathematics, was the most unexperienced of all participants. He had a strong background in C programming in Linux environment, but no practice in Java and modern development tools. The lack of experience imposed certain difficulties to execute the experiment, due to its demand for a minimum set of skills. The training given to the volunteer had a wider content to cover the bare minimum requirements to perform the tasks. An in-depth training in topics such as Java programming, Eclipse IDE, and debugging was out of scope.

The first task given to the user was to investigate the XStream framework. After fifty five minutes, he ended the first task and claimed that he had found the erroneous statement. Though the statement chosen was a part of the path of calls that lead to the defect, it was not the correct location.

The following task, Commons Math, had a duration of twenty two minutes. Again, the participant did not spot the location, but this time he was able to point the class where the defect happened.

In general, the participant had a difficult time to perform the tasks. He reported problems with basic actions, such as starting a debug session or adding a breakpoint in the source code. Furthermore, there was the immediate barrier imposed by the need to debug code written in a programming language that was not completely understood by him.

This is, in our opinion, a positive result. A developer with no Java, JUnit, Eclipse IDE, or industrial experience was able to find the trail leading to both defects using the plug-in.

To investigate the XStream (Table 13), the participant spent almost three minutes exploring the forest; he clicked eleven cacti and nine branches, and selected elements of the roadmap thirteen times; he also had to reset the view eight times. Total time spent debugging: almost six minutes. Continuing the experiment, the volunteer dedicated almost one minute to explore the Commons Math forest, picking one cactus and twelve branches; six elements of the roadmap were selected, and two resets of the view fired. The time dedicated debugging the code is very similar to the previous task, six minutes.

Table 13 *– Experiment # 1 summary*

	XStream	**Commons Math**
	CodeForest	
Time exploring	02:35	00:48
Movements	1144	374
	Debugging	
Time spent	06:25	05:52
Events	1	1
	Breakpoints	
Added	6	6
Removed	4	4
	Roadmap	
Events	13	6
	Trimmer	
Adjustments (text and score)	1	0
	Elements	
Cacti	11	1
Branches	9	12
Thorns	0	0
	Reset	
Events	8	2
	Task	
Duration	42:00	24:00
Defect found?	No	No

From the participant's answers (Table 17 - Appendix D), overall experience was consistently good, with few points of attention. According to him, the way the observer moves amongst cacti in the forest is neither good nor bad, same rank given to the quality of information structure.

The modifications suggested by the volunteer were: a keyboard shortcut that automatically changes the focus to the forest; a way to turn the interactions between mouse and keyboard more seamless, making them more interchangeable amongst themselves.

6.3.2 Experiment # 2

Participant #2 was an undergraduate student in Information Systems. He had good, though not industrial, coding skills in Java and JUnit. His ability to debug software using the Eclipse IDE needs further improvement, due to the extensive use of "print" calls to inspect elements and to follow the software's execution flow. This technique brings

significant risks to the debugging process, because it introduces unnecessary code to the existing codebase. Worst, this can change the way the software behaves and introduce new defects.

The experiment started with the investigation of the Commons Math framework. After forty two minutes, the volunteer claimed he had solved the first task. Just as his predecessor, the spotted location was not the one causing the defect. To complete this task, the volunteer spent forty seconds exploring the forest; he clicked four thorns, selected elements of the roadmap twenty two times and debugged for two minutes.

As for the second task, by the end of thirty seven minutes, the candidate was not able to point the location of the defect. He interacted with the XStream forest during one and a half minute, clicked six cacti and twelve thorns; he also selected elements of the roadmap twenty three times, and fired one reset of the view. His debug sessions lasted ten minutes. Usage statistics obtained from instrumentation logs are available on Table 14.

Table 14 *– Experiment # 2 summary*

	Commons Math	**XStream**
	CodeForest	
Time exploring	00:37	01:14
Movements	283	370
	Debugging	
Time spent	01:36	10:04
Events	6	13
	Breakpoints	
Added	7	9
Removed	6	9
	Roadmap	
Events	22	23
	Trimmer	
Adjustments (text and score)	0	0
	Elements	
Cacti	0	6
Branches	0	0
Thorns	4	12
	Reset	
Events	0	1
	Task	
Duration	41:00	35:00
Defect found?	No	No

From the participant's answers (Table 18 - Appendix D), overall experience was

good, although the main features provided by CodeForest (positioning, filtering and roadmap) were considered neutral, i.e, they did not help neither got in the way of the participant during the investigation. The low ratio between forest navigation and experiment duration suggests that the volunteer possibly ignored the plug-in and took advantage of the roadmap to set a starting point for investigation; a location where he could place "print" statements in order to analyse the problem. He also emphasized in his feedback the need to make thorns easier to click.

6.3.3 Experiment # 3

Participant #3 had the largest development experience of all four volunteers. A Master's student with an Information Systems diploma; more than six years of experience with Java and the Eclipse IDE environment; between four and six years of professional Java experience, and JUnit framework. Not only he was able to spot the defects' exact location, but he also corrected them, making failing JUnit test cases pass again.

The first task, XStream, demanded twenty five minutes from the volunteer, the lowest time of all participants. He explored the forest for thirty seconds, selected six cacti, five branches and two thorns; selected three elements from the roadmap list, and debugged for seven minutes.

As for the Commons Math task, the time needed to complete it was even shorter than the previous task: fifteen minutes. The CodeForest was explored for about one minute; the participant selected four cacti, three branches and ten thorns; he selected fifteen roadmap items, and debugged for three minutes. These, and other data collected during the experiment are available on Table 15.

At the end of the experiment, the participant's opinion about the plug-in was very positive (Table 19 - Appendix D), even though he reported having some trouble moving amongst elements in the forest. He also considered the forest having low utility in the task of locating defects. We believe that this opinion is related to the strong experience possessed by the volunteer on writing and debugging software, which possibly makes the visualization to be a desirable, although non essential, clue to locate the defect.

Table 15 *– Experiment # 3 summary*

	XStream	**Commons Math**
	CodeForest	
Time exploring	00:28	00:53
Movements	542	527
	Debugging	
Time spent	07:36	02:38
Events	4	3
	Breakpoints	
Added	2	2
Removed	0	0
	Roadmap	
Events	3	15
	Trimmer	
Adjustments (text and score)	0	133
	Elements	
Cacti	6	4
Branches	5	3
Thorns	2	10
	Reset	
Events	0	0
	Task	
Duration	22:00	17:00
Defect found?	Yes	Yes

6.4 Answers

The results from the validation experiment were gathered together with the results from the other three experiments. This could never happen in a meticulous study, but in this research we chose to do it. The reason is that our intent was to perform an exploratory—instead of a quantitative—analysis of the tool. The goal was to have insights, gather opinions and discover new ways that could lead to improvements in the plug-in.

Appendix E contains charts that summarizes the answers provided to the questions presented to volunteers, described in Appendix C. Figures 38, 39, and 40 show the answers to the questions related to user experience, feature utilization and impressions about the tool. Figure 41 sums up the answers regarding the expertise level of each developer on each skill.

From the answers it was possible to infer that volunteers had a good experience

utilizing the tool. They manifested the same opinion when asked to evaluate presentation characteristics, such as font size and the quality of information structure. One aspect of the metaphor—painting and positioning branches according to their score—received divergent evaluations and requires further experimentation. Except for that, the rationale behind the metaphor—the representation of the codebase as a forest, lines of code as thorns, and classes as cacti—was well accepted by participants and received high scores.

A key feature, the navigation within the CodeForest and its elements, demands improvements. Volunteers reported troubles when they tried to explore the forest; this creates a barrier between the participant and the metaphor, affects the amount of time dedicated to navigation and their opinion about the helpfulness provided by the tool to locate defects. Despite navigation issues, functional and non-functional requirements as performance, easiness to perform actions, and the interaction with elements of the forest were well ranked by participants.

Volunteers took the roadmap as something not related to filter tools (trimmer and text search). From utilization logs and interviews, participants heavily relied on roadmap items to investigate the defect's location, without realizing that when they clicked something, both trimmer and text search parameters were updated based on roadmap information. This action trimmed off forest elements that did not match the constraints provided. They did not realize—which possibly justifies the low scores—that they were extensively filtering content by selecting roadmap items.

Finally, participants were unanimous to point that they considered visual debugging of programs a very important topic in the Software Engineering field of study. These answers contrasted with a related question, which inquired participants whether they would always utilize the plug-in when available. Some hypothesis arise from this contrast: the forest metaphor might be inadequate to the problem in question; or perhaps the metaphor is good enough, but its realization was poorly executed; or maybe our choice of defects was not sufficient to convince developers of the aid that could have been provided by the tool. These questions opens new paths for further research and possible ways to improve the tool.

6.5 Threats to validity

The fundamental question about the results of every research is: "How valid are they?". The validity is affected by four types of threats: external, internal, construct,

and conclusion. In this section, we analyze the threats to the validity of the exploratory experiment conducted in this research.

6.5.1 External validity

External validity refers to the ability to generalize the study findings to other people or settings. The findings in this research cannot be generalized beyond the group of people that took part in the experiment. This accomplishment would require more participants and a meticulous experimental design. For example: we could ask participants to debug with and without the CodeForest; or with and without certain tools embedded within the plug-in.

6.5.2 Conclusion validity

Conclusion validity is the degree to which conclusions we reach about relationships in our data are reasonable. The goal of our exploratory experiment was to assess how the features embedded within the CodeForest plug-in were perceived by the participants. This way, the summary extracted from the data of every debugging session captures the way participants utilized the tool. Furthermore, we asked them to fill a questionnaire in order to measure their opinion about the plug-in after utilizing it. Hence, it is possible to evaluate how participants utilized the CodeForest.

6.5.3 Internal validity

Internal validity is concerned with the extent to which the design of the study allows us to attribute causality to the intervention and to rule out potential alternative explanations. In our case, there are influences from other factors. The previous experience of a participant plays an important role in the way he or she utilizes the tool. Moreover, all volunteers are part of the same research group, the Software Analysis & Experimentation Group (SAEG).

6.5.4 Construct validity

Construct validity is concerned with the relationship between theory and observation; more specifically whether a test measures the intended construct. It examines the question: “Does the measure behave like the theory says a measure of that construct

should behave?". Our experiment intended to perform an exploratory study of the CodeForest. This way, there was no need to compare measurements, but only the utilization of the tool itself. Hence, we do not incur into effects of combination of treatments neither sub-representation of the experiment. However, participants may have taken an additional effort to utilize the tool, something that might have not been done during normal utilization.

6.6 Final remarks

This chapter described the steps taken to plan, execute and analyze the results from an exploratory experiment, which aimed at the validation of the CodeForest plug-in and its integration into the Eclipse IDE environment.

In order to do that, it was necessary to reimplement the proof-of-concept program (detailed in Chapter 4) as a plug-in of an Integrated Development Environment. In this research, we chose the Eclipse platform to be the hosting system of our plug-in; the motivation behind this choice, architectural details and technical decisions are explained in Chapter 5.

After each experiment, an interview was performed with each participant. We asked their opinion and did a brainstorm on further ways to improve the tool. We checked how they utilized the plug-in; whether or not they successfully completed the task, along with other thoughts about the whole process. All this data was individually analised (Section 6.3) and grouped into charts (Section 6.4).

We gathered together the results from the validation experiment and the remaining three experiments. As previously explained, we chose this approach not to conduct a meticulous experiment, but rather to evaluate the way developers utilize the plug-in and to seek ways of improving it. Of four participants, two of them were able to find the defect. It is possible that this outcome is related to the fact that both of them had professional experience in jobs that required debugging and correcting code that was not written by them. The other two participants—which did not have any professional experience—could not find the line of code that originated the defect, though they were able to point the correct class and a method that was part of the chain of invocation that caused the defect.

Both groups of participants heavily relied on the roadmap to investigate the defect, and spent a considerable amount of time exploring the CodeForest. Overall opinion

about the plug-in was quite positive, and few aspects of the software requires adjustments; for example, the way users explore the forest and the tactic of positioning methods according to their suspiciousness score.

There are threats to the validity of this research, though. One is that the experiment here described is exploratory, which limits the range of our findings; another is that all participants are part of the same research group, which could introduce bias into our results.

The following chapter summarizes this research and draws conclusions about it. We also point directions in which one could extend this work.

Chapter 7

Conclusions and Future Work

In this chapter, we present the summary of the results achieved, our contributions, and the future work.

7.1 Summary

Throughout the last three decades, several metaphors have been proposed and adapted with the purpose of displaying source code. In 1997, Jim Foley expressed his disappointment with the lack of impact of Software Visualization techniques (STASKO; BROWN; PRICE, 1997) used as a means to debug programs and systems. He had hope that, in the near future, these techniques would have had their apex, transforming the Software Engineering field. Our research revealed that, although several visualizations have been developed, most of them are poorly suited to the task of debugging.

This research approached the area of debugging due to its importance in our digital world. In a planet that is more interconnected by systems than the day before, delivering and maintaining software with few or no defects became a worldwide concern. Recent studies estimate the global cost of debugging in more that 312 billion of dollars a year (BRITTON L. JENG; KATZENELLENBOGEN, 2013). This number makes imperative that better debugging tools need to be developed.

Our goal is to design and implement a visualization metaphor capable of helping developers to investigate defects more effectively. For that task, we created the CodeForest metaphor, whose rationale is to display parts of the codebase that were executed by unit tests as elements of a forest.

The metaphor maps classes into cacti, methods into branches, and lines of code into thorns. The application of the heuristic developed by (SOUZA, 2012) associates every element with a score, ranging from zero (lowest suspicion) to one (highest suspicion). This score expresses the chance of one given element to contain (in the case of a class or a method) or to be the cause (a line of code) of a defect.

The CodeForest metaphor was implemented as a plug-in of the Eclipse Integrated Development Environment. This choice was motivated by the features provided by the platform—code editor, compiler, debugger, and unit-test runner—but also by its wide popularity amongst professional developers. This job was not trivial; it demanded decisions such as: the most adequate 3D engine, how to turn a concept into a tool to debug, which mechanisms of the hosting platform the plug-in should mix with, and what is the best strategy to pack and distribute it.

To validate our novel tool, we planned and executed an exploratory experiment. Four volunteers—all members of the SAEG research group—took part in the tests. The experiment was modeled as follows: two frameworks were presented to participants—Thoughtworks XStream and Apache Commons Math—both of them with one defect. After a short training session, volunteers would have a first round of sixty minutes and a second one of thirty minutes to spot the line that triggered the failure in JUnit tests. Participants were not obliged to use the plug-in, remaining free to investigate the frameworks using their technique of choice.

Every participant utilized the CodeForest; two of them—the ones with professional background—were able to spot the defect, whilst the other two volunteers did not complete the task. Users had a positive perception of the tool, reporting few complaints. According to them, the navigation through the elements of the forest did not flow as they expected to; they also reacted negatively to our strategy of setting the position of branches (methods) in a cactus. These branches were organized according to their score: the higher the score, the closer the branch is to the ground.

The experiment revealed that the tool is capable of aiding developers with and without previous experience. Users with low or no experience utilized the roadmap together with the virtual 3D environment to investigate the defect. More experienced users utilized the roadmap as a guide to narrow their search of which parts of the source code should be inspected.

This research is not free of threats to its validity, which can be of four kinds: external, internal, construct and conclusion. The threat to external validity is that our results cannot be generalized beyond the group of volunteers that took part in the experiment. The threat to internal validity comes from the fact that the experiment is influenced by the previous experience of participants; moreover they were all part of the same research group. Finally, we claim that there are no threats to the conclusion validity neither the construct validity, due to the nature of our experiment: an exploratory experiment. It

was not our intent to measure whether theory and observation behaved accordingly to what was expected (construct validity) nor to attribute causality (conclusion validity).

7.2 Contributions

This research gave origin to several contributions in the context of visualization of coverage based debugging information that are described below:

- A novel three-dimensional metaphor—CodeForest—which aims at representing suspiciousness data obtained from coverage based debugging techniques.
- An extensive literature review. This research compiled a thorough literature review of 2D and 3D methods to visualize software data. In this process, we extracted important aspects from the analyzed papers: the concept ("what, conceptually, are the authors trying to achieve?"), how the concept was implemented, which aspects are covered by the research (structure, behavior, and evolution), the number of dimensions (2D or 3D), and the target audience (developers, stakeholders or system administrators).
- A fully functional prototype. The prototype described in Chapter 4, besides being used to validate our rationale about the metaphor, was also utilized by other researchers of the Software Analysis & Experimentation Group (SAEG) to perform experiments and produce results.
- A plug-in targeted to the Eclipse IDE that utilizes several of the underlying platform resources, which is already being extended to serve other purposes, mainly as the foundation of a system to explore other debugging techniques.
- An exploratory experiment, in which we assessed the adequacy of our new CodeForest metaphor in the activity of investigating defects.

7.3 Future work

Despite of the promising results, there is room for improvement. CodeForest requires a stronger validation, done preferably via a quantitative experiment. In this new experiment, participants should have different profiles and varied knowledge in Java, so they could be separated into different groups. The training session that happens before

the experiment needs improvement; it could be divided into, at least, four sessions: one for Eclipse, other for debugging techniques, another for JUnit and a final session on the CodeForest plug-in.

To improve filtering capabilities, the codebase could be indexed using an information retrieval framework such as Apache Lucene (LUCENE, 2005). Amongst its capabilities, Lucene allows users to to perform many powerful query types: phrase queries, wildcard queries, proximity queries, range queries, etc. This feature could empower users to perform queries such as "throws exception and method not static", which we believe would help developers to reach the defect location in less time. The tool could also use a faster 3D engine and better instrumentation logs, both of them required before engaging into another round of experiments.

As a concluding remark, the plug-in will reach its full effectiveness after the integration with external dependencies (Section 5.6) is completed. This will enable developers to perform changes in the source code and immediately get the feedback from a new forest, generated by the ICD/InSS after unit tests are run.

References

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. C. van. An observation-based model for fault localization. In: *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. New York, NY, USA: ACM, 2008a. (WODA '08), p. 64–70. ISBN 978-1-60558-054-8. Disponível em: <http://doi.acm.org/10.1145/1401827.1401841>.

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. J. V. An evaluation of similarity coefficients for software fault localization. In: IEEE. *Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on*. [S.l.], 2006. p. 39–46.

ABREU, R.; ZOETEWEIJ, P.; GEMUND, A. van. On the accuracy of spectrum-based fault localization. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*. [s.n.], 2007. p. 89–98. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4344104>.

ADMINISTRATION, E. I. *Short-Term Energy Outlook*. February 2012. `http://www.eei.org/ourissues/EnergyEfficiency/Documents/Prices_graph_2012.pdf`.

AGRAWAL, H.; HORGAN, J. R.; LONDON, S.; WONG, W. E. Fault localization using execution slices and dataflow tests. In: *Proceedings 6th International Symposium on Software Reliability Engineering (ISSRE 1995)*. [S.l.: s.n.], 1995. p. 143–151.

ALBAUM, G. The likert scale revisited. *Journal-Market research society*, MARKET RESEARCH SOC, v. 39, p. 331–348, 1997.

ALVES, A. D. C. *Osgi in Depth*. Manning Publications Company, 2011. (Manning Pubs Co Series). ISBN 9781935182177. Disponível em: <http://books.google.com.br-/books?id=RKxIYgEACAAJ>.

AMAZON.COM. *Books - Computers & Technology - Programming - Graphics & Multimedia - "java3d"*. January 2014. `http://www.amazon.com/s/ref=sr_nr_n_1?rh=n%3A3916%2Ck%3Ajava3d&keywords=java3d`.

AMAZON.COM. *Books - Computers & Technology - Programming - Languages & Tools - "eclipse"*. January 2014. `http://www.amazon.com/s/ref=sr_nr_n_0?rh=n%3A283155%2Cn%3A5%2Cn%3A3952%2Ck%3Aeclipse&keywords=eclipse`.

AMAZON.COM. *Books - Computers & Technology - Programming - Languages & Tools - "intellij"*. January 2014. `http://www.amazon.com/s/ref=nb_sb_noss_2?url=node%3D3952&field-keywords=intellij&rh=n%3A283155%2Cn%3A5%2Cn%3A3839%2Cn%3A3952%2Ck%3Aintellij`.

AMAZON.COM. *Books - Computers & Technology - Programming - Languages & Tools - "netbeans"*. January 2014. http://www.amazon.com/s/ref=nb_sb_noss_1?url=node%3D3952&field-keywords=netbeans&rh=n%3A283155%2Cn%3A5%2Cn%3A3839%2Cn%3A3952%2Ck%3Anetbeans.

BALZER, M.; DEUSSEN, O.; LEWERENTZ, C. Voronoi treemaps for the visualization of software metrics. In: ACM. *Proceedings of the 2005 ACM symposium on Software visualization*. [S.l.], 2005. p. 165–172.

BECK, K.; BEEDLE, M.; BENNEKUM, A. van; COCKBURN, A.; CUNNINGHAM, W.; FOWLER, M.; GRENNING, J.; HIGHSMITH, J.; HUNT, A.; JEFFRIES, R. et al. The agile manifesto. *The agile alliance*, v. 200, n. 1, 2001.

BLEWITT, A. *Eclipse 4 Plug-in Development by Example Beginner's Guide*. Packt Publishing, 2013. (Open source : community experience distilled). ISBN 9781782160335. Disponível em: <http://books.google.com.br/books?id=6uimZjojweIC>.

BRITTON L. JENG, G. C. P. C. T.; KATZENELLENBOGEN, T. Reversible debugging software. In: *Technical report*. University of Cambridge, 2013. Disponível em: <goo.gl/KzJd3g>.

BROWN, A.; WILSON, G. *The Architecture Of Open Source Applications*. lulu.com, 2011. 432+ p. Paperback. ISBN 1257638017. Disponível em: <http://www.aosabook-.org/en/>.

BUXTON, J.; RANDELL, B. et al. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee*. [S.l.]: NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

CAMPOS, J.; RIBOIRA, A.; PEREZ, A.; ABREU, R. Gzoltar: an eclipse plug-in for testing and debugging. In: ACM. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. [S.l.], 2012. p. 378–381.

CAUDWELL, A. H. Gource: visualizing software version control history. In: ACM. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. [S.l.], 2010. p. 73–74.

CENTER, N. S. W. *IMDb Alternative Interfaces*. September 1947. http://www.history.navy.mil/photos/images/h96000/h96566kc.htm.

CHAIM, M. L.; ARAUJO, R. P. A. de. An efficient bitwise algorithm for intra-procedural data-flow testing coverage. *Information Processing Letters*, v. 113, n. 8, p. 293–300, April 2013.

CHAIM, M. L.; MALDONADO, J. C.; JINO, M. A debugging strategy based on the requirements of testing. *Journal of Software Maintenance*, v. 16, n. 4-5, p. 277–308, 2004.

CLAYBERG, E.; RUBEL, D. *eclipse Plug-ins*. [S.l.]: Pearson Education, 2008.

COMMONS, A. *Math-Commons Math: The Apache Commons Mathematics Library*. 2013.

COMMUNICATIONS; COMMISSION, I. T. *Organizational Structure*. May 2013. http://www.citc.gov.sa/English/AboutUs/Pages/OrgStructure.aspx.

CORPORATION, O. *NetBeans IDE*. 2014. http://netbeans.org.

D'ANJOU, J. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2005. ISBN 9780321305022. Disponível em: <http://books.google.com.br-/books?id=6Ob1ANNVcXcC>.

DICKINSON, W.; LEON, D.; PODGURSKI, A. Finding failures by cluster analysis of execution profiles. In: *Proceedings of the 23rd International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2001. (ICSE '01), p. 339–348. ISBN 0-7695-1050-7. Disponível em: <http://dl.acm.org/citation-.cfm?id=381473.381509>.

DIEHL, S. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007. ISBN 3540465049.

DIX, A. *Human-computer interaction*. [S.l.]: Prentice hall, 2004.

EICK, S.; STEFFEN, J.; JR, E. S. Seesoft-a tool for visualizing line oriented software statistics. *Software Engineering, IEEE Transactions on*, IEEE, v. 18, n. 11, p. 957–968, 1992.

ELBAUM, S.; GABLE, D.; ROTHERMEL, G. The impact of software evolution on code coverage information. In: *Proceedings of the International Conference on Software Maintenance*. [S.l.]: IEEE, 2001. (ICSM '01), p. 170–179.

ERRA, U.; SCANNIELLO, G. Towards the visualization of software systems as 3d forests: the codetrees environment. In: ACM. *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. [S.l.], 2012. p. 981–988.

FEW, S. *Information dashboard design*. [S.l.]: O'Reilly, 2006.

FOUNDATION, E. *Eclipse IDE*. 2014. http://eclipse.org.

FOUNDATION, E. *Platform Plug-in Developer Guide - Programmer's Guide - Platform architecture*. January 2014. http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2Farch.htm.

FRIENDLY, M. A brief history of data visualization. *Handbook of data visualization*, Springer, p. 15–56, 2008.

FRIENDLY, M.; DENIS, D. J. Milestones in the history of thematic cartography, statistical graphics, and data visualization. *Accessed: March*, v. 18, p. 2010, 2001.

FURMANKIEWICZ, J. *Eclipse, NetBeans, and IntelliJ: Assesing the Survivors of the Java IDE Wars*. 2008.

GAMMA, E.; BECK, K. *JUnit*. 2006.

GEER, D. Eclipse becomes the dominant java ide. *Computer*, IEEE, v. 38, n. 7, p. 16–18, 2005.

GONZÁLEZ, A. *Automatic Error Detection Techniques based on Dynamic Invariants*. Dissertação (Mestrado) — Delft University of Technology, 2007. Disponível em: <www.st.ewi.tudelft.nl/~albgonz/pub/mscthesis.pdf>.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G. *Java (TM) Language Specification, The (Java (Addison-Wesley))*. [S.l.]: Addison-Wesley Professional, 2005.

GRAHAM, H.; YANG, H.; BERRIGAN, R. A solar system metaphor for 3d visualisation of object oriented software metrics. In: AUSTRALIAN COMPUTER SOCIETY, INC. *Proceedings of the 2004 Australasian symposium on Information Visualisation-Volume 35*. [S.l.], 2004. p. 53–59.

GROVE, D.; DEFOUW, G.; DEAN, J.; CHAMBERS, C. Call graph construction in object-oriented languages. In: ACM. *ACM SIGPLAN Notices*. [S.l.], 1997. v. 32, n. 10, p. 108–124.

HALL, R.; PAULS, K.; MCCULLOCH, S. *OSGi in Action: Creating Modular Applications in Java*. Manning Publications Company, 2011. (In Action). ISBN 9781933988917. Disponível em: <http://books.google.com.br/books?id=y3lPPgAACAAJ>.

HARROLD, M. J.; ROTHERMEL, G.; WU, R.; YI, L. An empirical investigation of program spectra. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 33, n. 7, p. 83–90, jul. 1998. ISSN 0362-1340. Disponível em: <http://doi.acm.org/10.1145/277633.277647>.

HEALEY, C. G.; BOOTH, K. S.; ENNS, J. T. High-speed visual estimation using preattentive processing. *ACM Transactions on Computer-Human Interaction (TOCHI)*, ACM, v. 3, n. 2, p. 107–135, 1996.

HELANDER, M. G.; LANDAUER, T. K.; PRABHU, P. V. *Handbook of human-computer interaction*. [S.l.]: North Holland, 1997.

HOFFMANN, M. R. *Eclemma-java code coverage for eclipse*. 2009.

HUMPHREY, W. S. *Bugs or Defects?* May 2013. `http://www.sei.cmu.edu/library/abstracts/news-at-sei/wattsmar99.cfm`.

IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. 1–84, 1990.

IEEE. Systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, p. 1–418, 2010.

INC., T. *Thoughtworks Anthology, Volume 2: More Essays on Software Technology and Innovation*. Pragmatic Bookshelf, 2012. ISBN 9781937785000. Disponível em: <http://books.google.com.br/books?id=0hRvMAEACAAJ>.

INDURKHYA, B. The thesis that all knowledge is metaphorical and meanings of metaphor. *Metaphor and Symbol*, Taylor & Francis, v. 9, n. 1, p. 61–73, 1994.

JETBRAINS. *IntelliJ IDEA IDE*. 2014. `http://www.jetbrains.com/idea/`.

JOHNSON, R.; RIVAIT, J.; BARR, A. *U.S. 2012 election results*. November 2012. `http://news.nationalpost.com/2012/11/07/graphic-u-s-2012-election-results/`.

JONES, J.; HARROLD, M.; STASKO, J. Visualization of test information to assist fault localization. In: ACM. *Proceedings of the 24th international conference on Software engineering*. [S.l.], 2002. p. 467–477.

JONES, J. A.; BOWRING, J. F.; HARROLD, M. J. Debugging in parallel. In: *Proceedings of the 2007 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2007. (ISSTA '07), p. 16–26. ISBN 978-1-59593-734-6. Disponível em: <http://doi.acm.org/10.1145/1273463.1273468>.

KABANOV, J. *The 2011 Turnaround, Container and Tools Pre-Report Snapshot?* May 2012. `http://zeroturnaround.com/rebellabs/developer-productivity-report-2012-java-tools-tech-devs-and-data/4/`.

KLEIBERG, E.; WETERING, H. V. D.; WIJK, J. V. Botanical visualization of huge hierarchies. In: *Proceedings of the IEEE Symposium on Information Visualization*. [S.l.: s.n.], 2001. p. 87.

KNIGHT, C.; MUNRO, M. Virtual but visible software. In: IEEE. *Information Visualization, 2000. Proceedings. IEEE International Conference on*. [S.l.], 2000. p. 198–205.

LAKOFF, G.; JOHNSON, M. *Metaphors we live by*. [S.l.]: Chicago London, 1980.

LARAMEE, R. S. How to read a visualization research paper: Extracting the essentials. *Computer Graphics and Applications, IEEE*, IEEE, v. 31, n. 3, p. 78–82, 2011.

LUCENE, A. *Apache software foundation*. 2005.

MACKINLAY, J. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics (TOG)*, ACM, v. 5, n. 2, p. 110–141, 1986.

MYATT, G.; JOHNSON, W. *Making sense of data 2*. [S.l.]: Wiley, 2009.

MYERS, B. A. A brief history of human-computer interaction technology. *interactions*, ACM, v. 5, n. 2, p. 44–54, 1998.

ORSO, A.; JONES, J.; HARROLD, M.; STASKO, J. Gammatella: Visualization of program-execution data for deployed software. In: IEEE COMPUTER SOCIETY. *Proceedings of the 26th International Conference on Software Engineering*. [S.l.], 2004. p. 699–700.

PANAS, T.; BERRIGAN, R.; GRUNDY, J. A 3d metaphor for software production visualization. In: IEEE. *Information Visualization, 2003. IV 2003. Proceedings. Seventh International Conference on*. [S.l.], 2003. p. 314–319.

PARNIN, C.; ORSO, A. Are automated debugging techniques actually helping programmers? In: ACM. *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. [S.l.], 2011. p. 199–209.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. *IEEE Trans. Software Eng.*, v. 11, n. 4, p. 367–375, 1985.

REINHARD, E.; KHAN, E. A.; AKYUZ, A. O. *Color Imaging: Fundamentals and Applications*. Taylor & Francis, 2008. (Ak Peters Series). ISBN 9781568813448. Disponível em: <http://books.google.com.br/books?id=79-V0_TElg4C>.

REISS, S. P.; BOTT, J. N.; LAVIOLA, J. Code bubbles: a practical working-set programming environment. In: IEEE. *Software Engineering (ICSE), 2012 34th International Conference on*. [S.l.], 2012. p. 1411–1414.

RENIERIS, M.; REISS, S. P. Fault localization with nearest neighbor queries. *18th IEEE International Conference on Automated Software Engineering 2003 Proceedings*, IEEE Comput. Soc, v. 0, p. 30–39, 2003. Disponível em: <http://ieeexplore.ieee.org/lpdocs-/epic03/wrapper.htm?arnumber=1240292>.

RYDER, B. G. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, IEEE, n. 3, p. 216–226, 1979.

SANTELICES, R. A.; JONES, J. A.; YU, Y.; HARROLD, M. J. Lightweight fault-localization using multiple coverage types. In: *ICSE*. [S.l.]: IEEE, 2009. p. 56–66. ISBN 978-1-4244-3452-7.

SHNEIDERMAN, B. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Transactions on graphics (TOG)*, ACM, v. 11, n. 1, p. 92–99, 1992.

SHNEIDERMAN, B. The eyes have it: A task by data type taxonomy for information visualizations. In: IEEE. *Visual Languages, 1996. Proceedings., IEEE Symposium on*. [S.l.], 1996. p. 336–343.

SOUZA, H. A. d. *Depuração de programas baseada em cobertura de integração*. Dissertação (Mestrado) — Escola de Artes, Ciências e Humanidades — Universidade de São Paulo, 2012.

SOUZA, H. A. de; CHAIM, M. L. Adding context to fault localization with integration coverage. In: *Proceeding of the 8th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013*. [S.l.]: IEEE, 2013. p. 628–633.

SOWIZRAL, H. A.; DEERING, M. F. The java 3d api and virtual reality. *Computer Graphics and Applications, IEEE*, IEEE, v. 19, n. 3, p. 12–15, 1999.

SOWIZRAL, K.; RUSHFORTH, K.; SOWIZRAL, H. *The Java 3D API Specification*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 1997.

STASKO, J. T.; BROWN, M. H.; PRICE, B. A. *Software visualization*. [S.l.]: MIT press, 1997.

TAFT, D. *Eclipse Luna Coordinated Release Train Ships*. June 2014. `http://www.eweek.com/developer/eclipse-luna-coordinated-release-train-ships.html`.

TREISMAN, A.; GORMICAN, S. Feature analysis in early vision: evidence from search asymmetries. *Psychological review*, American Psychological Association, v. 95, n. 1, p. 15, 1988.

TUFTE, E. R.; GRAVES-MORRIS, P. *The visual display of quantitative information*. [S.l.]: Graphics press Cheshire, CT, 1983.

WALNES, J.; SCHAIBLE, J.; TALEVI, M.; SILVEIRA, G. et al. Xstream. *URL: http://xstream.codehaus.org*, 2005.

WANDELL, B. A. *Foundations of vision.* [S.l.]: Sinauer Associates, 1995.

WARE, C. *Visual thinking for design.* [S.l.]: Morgan Kaufmann Pub, 2008.

WETTEL, R.; LANZA, M. Program comprehension through software habitability. In: IEEE. *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on.* [S.l.], 2007. p. 231–240.

WIJK, J. J. V.; WETERING, H. Van de. Cushion treemaps: Visualization of hierarchical information. In: IEEE. *Information Visualization, 1999.(Info Vis' 99) Proceedings. 1999 IEEE Symposium on.* [S.l.], 1999. p. 73–78.

WONG, W.; QI, Y.; ZHAO, L.; CAI, K.-Y. Effective fault localization using code coverage. In: *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International.* [s.n.], 2007. v. 1, p. 449–456. ISSN 0730-3157. Disponível em: <http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4291037>.

WOTAWA, F.; STUMPTNER, M.; MAYER, W. Model-based debugging or how to diagnose programs automatically. In: HENDTLASS, T.; ALI, M. (Ed.). *Developments in Applied Artificial Intelligence.* Springer Berlin / Heidelberg, 2002, (Lecture Notes in Computer Science, v. 2358). p. 243–257. ISBN 978-3-540-43781-9. 10.1007/3-540-48035-8_72. Disponível em: <http://dx.doi.org/10.1007/3-540-48035-8_72>.

YOSHIMURA, K.; MIBE, R. Visualizing code clone outbreak: An industrial case study. In: IEEE. *Software Clones (IWSC), 2012 6th International Workshop on.* [S.l.], 2012. p. 96–97.

ZANINOTTO, F. *CodeFlower Source code visualization.* April 2013. `http://redotheweb.com/CodeFlower/`.

ZELLER, A. Isolating cause-effect chains from computer programs. In: *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE).* [S.l.: s.n.], 2002. p. 1–10.

ZELLER, A. *Why programs fail: a guide to systematic debugging.* [S.l.]: Morgan Kaufmann, 2009.

ZHANG, Z.; CHAN, W.; TSE, T.; HU, P.; WANG, X. Is non-parametric hypothesis testing model robust for statistical fault localization? *Information and Software Technology*, Elsevier, v. 51, n. 11, p. 1573–1585, 2009.

APPENDIX A

Informed Consent Form

Título do Projeto e Objetivo:

Avaliação da ferramenta "CodeForest", parte integrante do projeto de mestrado "Coverage Based Debugging Visualization" desenvolvido na Escola de Artes, Ciências e Humanidades (EACH) da Universidade de São Paulo (USP).

Objetivos:

Proposta:	Analisar	A ferramenta CodeForest do ponto de vista funcional e suas características de usabilidade.
	Com propósito de	Avaliar se a ferramenta é capaz de auxiliar programadores em tarefas de identificação de linhas de código com defeito, bem como suas características de usabilidade.
Perspectiva:	Com respeito	À utilização da ferramenta e aos possíveis benefícios obtidos na tarefa de identificação de linhas de código com defeito.
	Do ponto de vista de	Programadores Java.

Procedimentos:

O procedimento do experimento é o que segue. Primeiramente haverá uma explicação de 30 minutos sobre a ferramenta e o seu funcionamento básico. A seguir, dois arcabouços para manipulação de XML serão brevemente apresentados, Thoughtworks XStream e Apache Commons-Math. Na sequência, os programadores terão 30 minutos para encontrar a linha de código que contém o defeito semeado utilizando ambiente de programação Eclilpse no qual a ferramenta CodeForest está instalada. Em seguida, haverá a aplicação de formulários em que cada participante fará a avaliação da ferramenta do ponto de vista funcional, de usabilidade e sua utilidade na tarefa proposta.

Confidencialidade:

Todas as informações coletadas no experimento são confidenciais e meu nome não será identificado em momento algum. Confirmo que sou maior de 18 anos e que desejo participar do experimento conduzido pelos professores Marcos Lordello Chaim e João Luiz Bernardes Junior e pelo mestrando Danilo Mutti.

Eu entendo que não serei remunerado pessoalmente por participar dessa avaliação e que os dados coletados poderão ser usados em publicações científicas. Eu compreendo que ficarei livre para fazer perguntas ou para deixar de participar da avaliação a qualquer momento sem sofrer penalidades.

Eu, _____ aceito participar do experimento de acordo com o descrito acima.

Local: _____ Data: _____ Assinatura: _____

APPENDIX B

Training Slides

Coverage Based Debugging Visualization

Danilo Mutti
Orientador Prof. Dr. Marcos Lordello Chaim

Motivação

- Defeitos são parte inerente do processo de desenvolvimento
- É uma atividade
  - notoriamente difícil
  - que consome grande quantidade de tempo
- Paradigma atual
  - Baseado quase que exclusivamente no uso de breakpoints e variantes

Objetivos

- Propor uma metáfora visual tri-dimensional para representar informações de depuração em programas grandes.
- Fornecer um roteiro que auxilie na localização de defeitos baseado na posição e cor dos elementos da metáfora.
- Disponibilizar ferramentas de busca e filtragem de elementos específicos na metáfora.

Estado da Arte

- Na indústria
  - *Breakpoints* e funcionalidades baseadas neste recurso
  - *Step over*
  - *Step into*
  - *Single stepping*
  - Avaliação de funções
- Na academia
  - Depuração delta
  - *Slicing* estático/dinâmico
  - Depuração baseada em cobertura

CodeForest

- Conjunto de casos de teste automatizados utilizando a biblioteca junit é executado.
- Informações de cobertura de código são coletadas:
  - comandos executados por cada caso de teste.
- Depuração baseada em cobertura:
  - atribui um valor de suspeição a nós (conjuntos de comandos executados sequencialmente), métodos e classes.

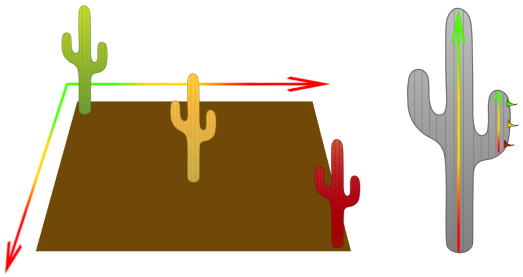
CodeForest

- Valor de suspeição é baseado na frequência de execução de um nó por casos de teste de sucesso e, principalmente, por casos de teste de falha.
- Atribui-se um *valor de suspeição* entre 0 e 1 para cada nó (conjunto de comandos executados sequencialmente), método e classe
- A suspeição de cada elemento (nó, método e classe) é o insumo para a visualização.

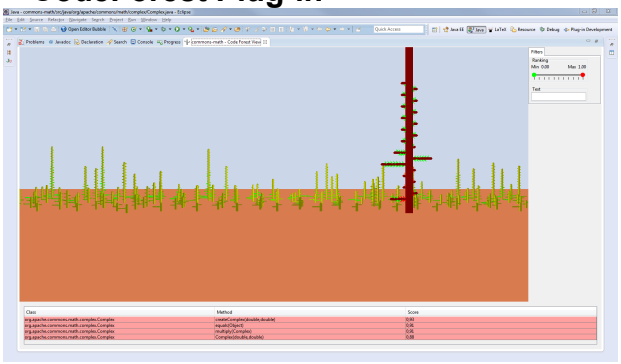
CodeForest

- Metáfora na qual
  - Árvores representam classes;
  - Galhos representam métodos;
  - Folhas representam linhas de código.
- Desafio
  - Exibir uma quantidade muito grande de dados em um espaço limitado.

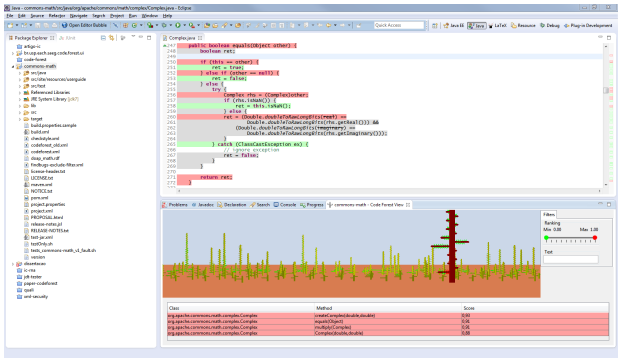
Metáfora CodeForest



CodeForest Plug-in



CodeForest Plug-in



Manual de Instruções

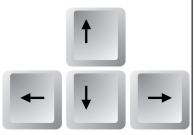
- Fazer a análise do projeto
  - Botão Direito > Code Forest > Perform analysis
- Linhas de código pintadas de acordo com o grau de suspeição
  - Alto
  - Médio Alto
  - Médio Baixo
  - Baixo
  - N/A
- Visualização da Floresta
  - Botão Direito > Code Forest > View as Code Forest

Manual de Instruções

- Trimmer
  - Ajusta o filtro de suspeição para os valores mínimos e máximos escolhidos.
- Filtro de texto
  - Deixa visíveis somente os elementos que possuem o texto digitado;
  - Independente de caixa alta ou baixa.
- Utilizados em conjunto "AND"
- Roteiro de busca baseado nos métodos mais suspeitos.

Manual de Instruções

- Movimentação para frente, para trás e para os lados com as setas.
- Controles especiais
  - Command (Mac) / Control (Win) - Escada
  - Shift - Pescoço
- Reset da view
  - tecla =



Manual de Instruções - Ferramentas

- Clique (esquerdo) em qualquer elemento da Floresta abre o elemento escolhido
  - Tronco - 1a linha da classe;
  - Galho - 1a linha do método;
  - Folha - linha de código.
- Clique (esquerdo) em qualquer item do script
  - Filtro de texto pelo nome do método;
  - Trimmer entre o grau de suspeição do método (máx) e um limiar mínimo.

Commons Math

- commons.apache.org/proper/commons-math
  - Biblioteca matemática leve, contida com componentes estatísticos.
  - Contém componentes pequenos e facilmente integráveis.
- org.apache.commons.math3.stat.StatUtils
  - mean(double[] values)
  - variance(double[] values)
  - sumSq(double[] values)

XStream

- xstream.codehaus.org
  - Uma biblioteca simples para serializar objetos para XML e de XML para objetos.
- XStream xstream = new XStream();
  - Person joe = new Person("Joe");
  - String xml = xstream.toXML(joe);
  - Person newJoe = (Person) xstream.fromXML(xml);

Demo da CodeForest

- CodeForest é baseada em informação de cobertura coletada a partir de testes automatizados com o JUnit.
- Está informação já foi coletada por meio das ferramentas InSS (*Instrumentation Strategies Simulator*) e DCI (Depuração baseada em Cobertura de Integração).
- O programador pode, se quiser, utilizar o Plugin da CodeForest para localizar o defeito.
- Ele é livre para usar a CodeForest com os recursos para depuração do Eclipse ou somente esses recursos, se assim o desejar.

Conclusões

- O que está sendo avaliado é o processo de depuração e não o participante do experimento.
- Você é livre para abandonar o experimento no momento que quiser, sem qualquer prejuízo.
- As informações coletadas no experimento poderão ser utilizadas para publicações científicas, mas seu anonimato é garantido.

APPENDIX C

Evaluation Form

1) O tamanho e a forma das fontes são muito agradáveis.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

2) A representação do código como uma floresta é útil para identificar trechos de código suspeitos de conter defeitos.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

3) É fácil se movimentar entre os elementos da floresta.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

4) É fácil selecionar um elemento da floresta.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

5) É fácil acessar o código-fonte.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

6) As informações exibidas são claras e organizadas adequadamente.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

7) O desempenho do plugin é plenamente adequado à depuração de programas.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

8) As funções do plug-in são executadas facilmente.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

9) Pintar e posicionar as linhas de código (espinhos) de acordo com o grau de suspeição ajuda a depurar.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

10) Posicionar os métodos (galhos) de acordo com o grau de suspeição ajuda a depurar.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

11) Pintar e posicionar as classes (cactus) de acordo com o grau de suspeição ajuda a depurar.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

12) A navegação pela floresta foi um recurso bastante utilizado para localização de defeitos da CodeForest.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

13) O roteiro de métodos foi um recurso bastante utilizado para localização de defeitos da CodeForest.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

14) A busca textual foi um recurso bastante utilizado para localização de defeitos da CodeForest.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

15) O "trimmer" foi um recurso bastante utilizado para localização de defeitos da CodeForest.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

16) Eu utilizarei o plug-in CodeForest sempre que ele estiver disponível no Eclipse.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

17) Precisei de pouco tempo para entender o plug-in.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

18) Depuração visual é importante para a Engenharia de Software.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

19) A experiência de utilização como um todo foi muito boa.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

20) Eu consegui localizar o defeito facilmente utilizando a floresta.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

21) Eu consegui localizar o defeito facilmente utilizando o roteiro.

	1	2	3	4	5	
Discordo totalmente	◯	◯	◯	◯	◯	Concordo totalmente

22) Qual o seu tempo de experiência com a linguagem Java?

◯ Nenhum
◯ Entre 0 e 2 anos
◯ Entre 2 e 4 anos
◯ Entre 4 e 6 anos
◯ Mais de 6 anos

23) Qual o seu tempo de experiência com o Eclipse IDE?

◯ Nenhum
◯ Entre 0 e 2 anos
◯ Entre 2 e 4 anos
◯ Entre 4 e 6 anos
◯ Mais de 6 anos

24) Qual o seu tempo de experiência utilizando o arcabouço JUnit?

◯ Nenhum
◯ Entre 0 e 2 anos
◯ Entre 2 e 4 anos
◯ Entre 4 e 6 anos
◯ Mais de 6 anos

25) Qual o seu tempo de experiência com as técnicas de depuração visual?

- ◯ Nenhum
- ◯ Entre 0 e 2 anos
- ◯ Entre 2 e 4 anos
- ◯ Entre 4 e 6 anos
- ◯ Mais de 6 anos

26) Qual o seu tempo de experiência com a linguagem Java em ambiente industrial?

- ◯ Nenhum
- ◯ Entre 0 e 2 anos
- ◯ Entre 2 e 4 anos
- ◯ Entre 4 e 6 anos
- ◯ Mais de 6 anos

APPENDIX D

Evaluation Form Answers

Table 16 – *Pilot experiment*

Question	Answer
Font size and shape are very pleasant	5
Representing source code as a forest is useful to spot suspicious lines of code	4
It is easy to move amongst forest elements	2
Information provided is clear and properly organized	5
Plug-in performance is completely adequate to debug programs	4
Plug-in functions are easily executed	5
Setting the color and the placement of lines of code (thorns) according to their suspiciousness value helps debugging	5
Setting the color and the placement of methods (branches) according to their suspiciousness value helps debugging	2
It took me a short time to understand the plug-in	3
Visual debugging is important to the Software Engineering field	5
Overall experience was really good	5
It was easy to locate the defect using the forest	3
It was easy to locate the defect using the roadmap	4
It is easy to access the source code	5
It is easy to pick a single element	5
Setting the color and the placement of classes (cacti) according to their suspiciousness value helps debugging	5
The navigation through the forest was highly used to locate the defect	3
The roadmap of methods was highly used to locate the defect	5
The text search was highly used to locate the defect	2
The trimmer was highly used to locate the defect	4
I will use the plug-in whenever available	4
What is your experience with Java?	2 to 4 years
What is your experience with the Eclipse IDE?	2 to 4 years
What is your experience with the JUnit Framework?	2 to 4 years
What is your experience with visual debugging techniques?	None
What is your experience with Java in an industrial environment?	0 to 2 years

Table 17 – *Experiment # 1*

Question	**Answer**
Font size and shape are very pleasant	**5**
Representing source code as a forest is useful to spot suspicious lines of code	**4**
It is easy to move amongst forest elements	**3**
Information provided is clear and properly organized	**3**
Plug-in performance is completely adequate to debug programs	**5**
Plug-in functions are easily executed	**4**
Setting the color and the placement of lines of code (thorns) according to their suspiciousness value helps debugging	**5**
Setting the color and the placement of methods (branches) according to their suspiciousness value helps debugging	**4**
It took me a short time to understand the plug-in	**5**
Visual debugging is important to the Software Engineering field	**5**
Overall experience was really good	**4**
It was easy to locate the defect using the forest	**4**
It was easy to locate the defect using the roadmap	**5**
It is easy to access the source code	**5**
It is easy to pick a single element	**4**
Setting the color and the placement of classes (cacti) according to their suspiciousness value helps debugging	**5**
The navigation through the forest was highly used to locate the defect	**4**
The roadmap of methods was highly used to locate the defect	**4**
The text search was highly used to locate the defect	**3**
The trimmer was highly used to locate the defect	**1**
I will use the plug-in whenever available	**3**
What is your experience with Java?	**None**
What is your experience with the Eclipse IDE?	**None**
What is your experience with the JUnit Framework?	**None**
What is your experience with visual debugging techniques?	**None**
What is your experience with Java in an industrial environment?	**None**

Table 18 – *Experiment # 2*

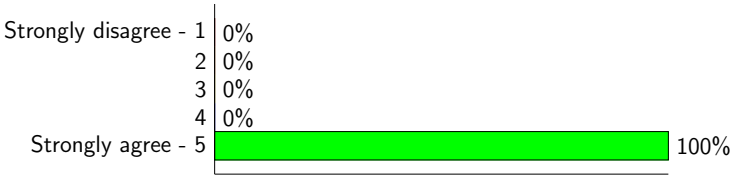
Question	**Answer**
Font size and shape are very pleasant	**5**
Representing source code as a forest is useful to spot suspicious lines of code	**4**
It is easy to move amongst forest elements	**3**
Information provided is clear and properly organized	**4**
Plug-in performance is completely adequate to debug programs	**4**
Plug-in functions are easily executed	**3**
Setting the color and the placement of lines of code (thorns) according to their suspiciousness value helps debugging	**3**
Setting the color and the placement of methods (branches) according to their suspiciousness value helps debugging	**3**
It took me a short time to understand the plug-in	**4**
Visual debugging is important to the Software Engineering field	**4**
Overall experience was really good	**4**
It was easy to locate the defect using the forest	**2**
It was easy to locate the defect using the roadmap	**3**
It is easy to access the source code	**5**
It is easy to pick a single element	**4**
Setting the color and the placement of classes (cacti) according to their suspiciousness value helps debugging	**4**
The navigation through the forest was highly used to locate the defect	**4**
The roadmap of methods was highly used to locate the defect	**5**
The text search was highly used to locate the defect	**2**
The trimmer was highly used to locate the defect	**2**
I will use the plug-in whenever available	**3**
What is your experience with Java?	**2 to 4 years**
What is your experience with the Eclipse IDE?	**2 to 4 years**
What is your experience with the JUnit Framework?	**None**
What is your experience with visual debugging techniques?	**None**
What is your experience with Java in an industrial environment?	**None**

Table 19 – *Experiment # 3*

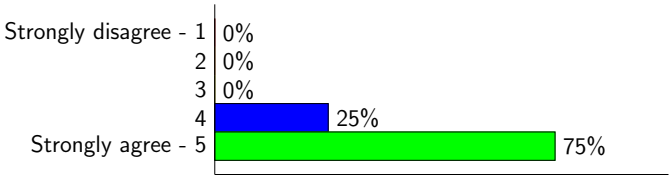
Question	**Answer**
Font size and shape are very pleasant	**5**
Representing source code as a forest is useful to spot suspicious lines of code	**5**
It is easy to move amongst forest elements	**3**
Information provided is clear and properly organized	**5**
Plug-in performance is completely adequate to debug programs	**5**
Plug-in functions are easily executed	**4**
Setting the color and the placement of lines of code (thorns) according to their suspiciousness value helps debugging	**4**
Setting the color and the placement of methods (branches) according to their suspiciousness value helps debugging	**5**
It took me a short time to understand the plug-in	**5**
Visual debugging is important to the Software Engineering field	**5**
Overall experience was really good	**5**
It was easy to locate the defect using the forest	**4**
It was easy to locate the defect using the roadmap	**4**
It is easy to access the source code	**4**
It is easy to pick a single element	**4**
Setting the color and the placement of classes (cacti) according to their suspiciousness value helps debugging	**4**
The navigation through the forest was highly used to locate the defect	**2**
The roadmap of methods was highly used to locate the defect	**5**
The text search was highly used to locate the defect	**1**
The trimmer was highly used to locate the defect	**2**
I will use the plug-in whenever available	**3**
What is your experience with Java?	**More than 6 years**
What is your experience with the Eclipse IDE?	**More than 6 years**
What is your experience with the JUnit Framework?	**More than 6 years**
What is your experience with visual debugging techniques?	**0 to 2 years**
What is your experience with Java in an industrial environment?	**4 to 6 years**

APPENDIX E

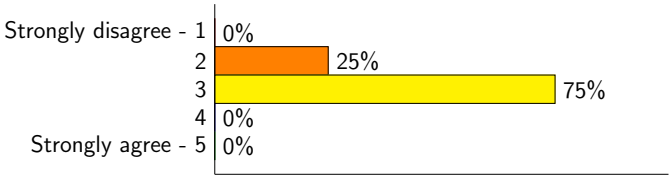
Evaluation Form Summary



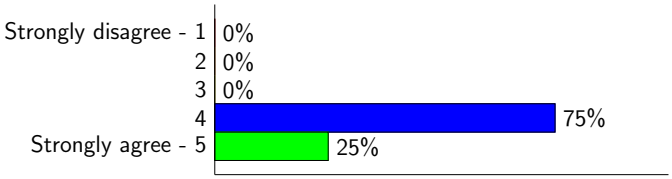
(a) Font size and shape are very pleasant



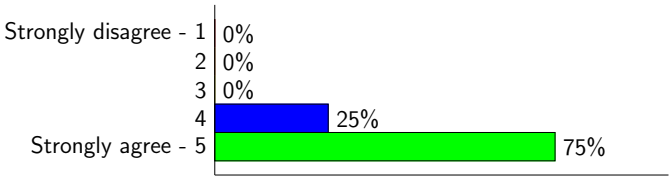
(b) Representing source code as a forest is useful to spot suspicious lines of code



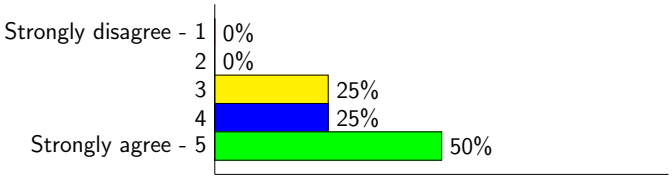
(c) It is easy to move amongst forest elements



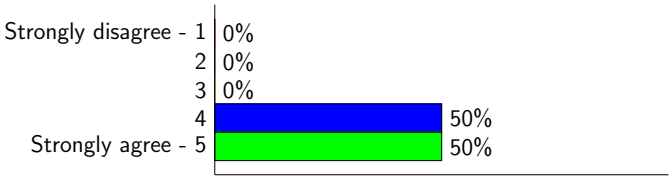
(d) It is easy to pick a single element



(e) It is easy to access the source code

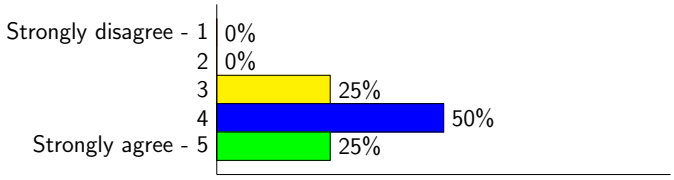


(f) Information provided is clear and properly organized

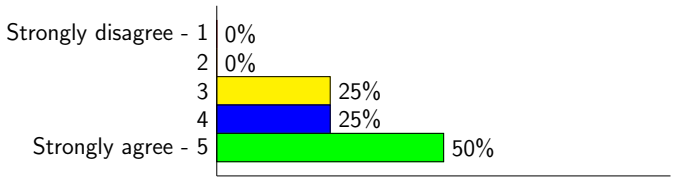


(g) Plug-in performance is completely adequate to debug programs

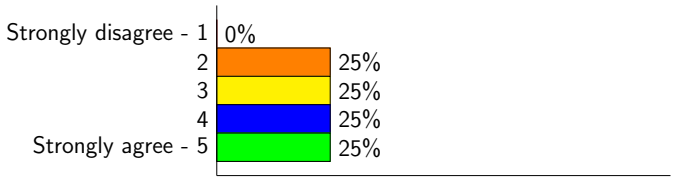
Figure 38 – *Answers 1 to 7*



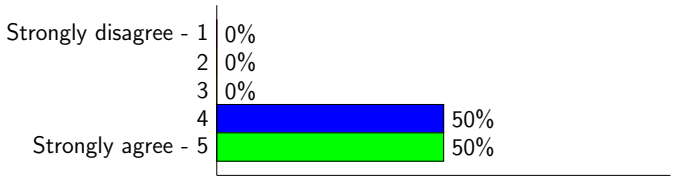
(a) Plug-in functions are easily executed



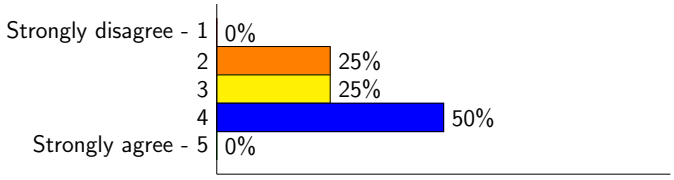
(b) Setting the color and the placement of lines of code (thorns) according to their suspiciousness value helps debugging



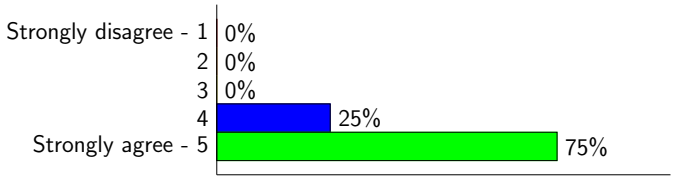
(c) Setting the color and the placement of methods (branches) according to their suspiciousness value helps debugging



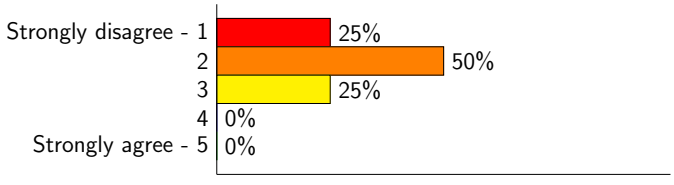
(d) Setting the color and the placement of classes (cacti) according to their suspiciousness value helps debugging



(e) The navigation through the forest was highly used to locate the defect

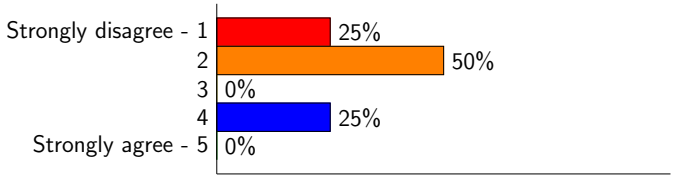


(f) The roadmap of methods was highly used to locate the defect

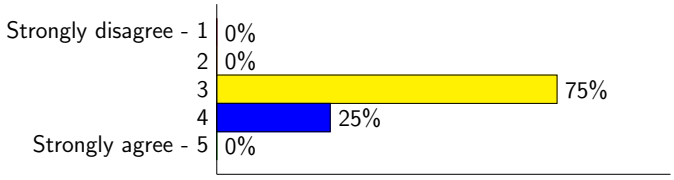


(g) The text search was highly used to locate the defect

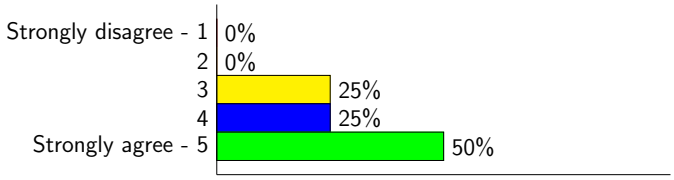
Figure 39 – *Answers 8 to 14*



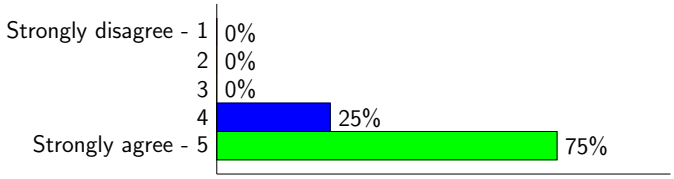
(a) The trimmer was highly used to locate the defect



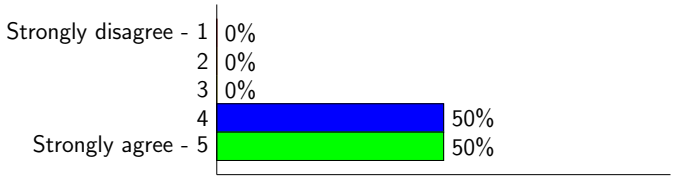
(b) I will use the plug-in whenever available



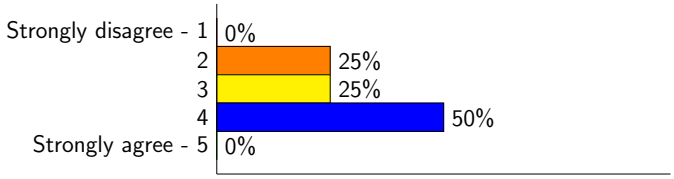
(c) It took me a short time to understand the plug-in



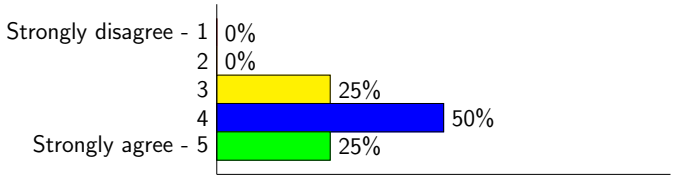
(d) Visual debugging is important to the Software Engineering field



(e) Overall experience was really good

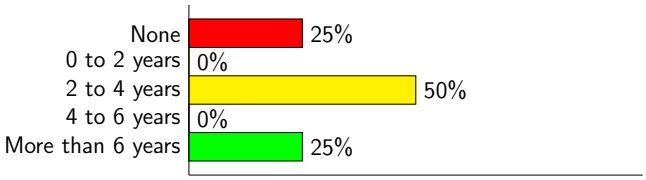


(f) It was easy to locate the defect using the forest

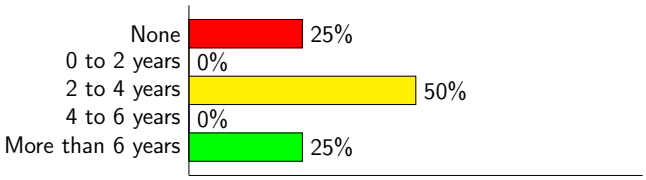


(g) It was easy to locate the defect using the roadmap

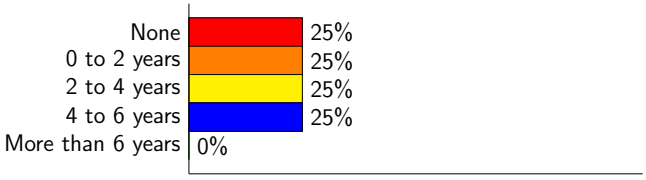
Figure 40 – *Answers 15 to 21*



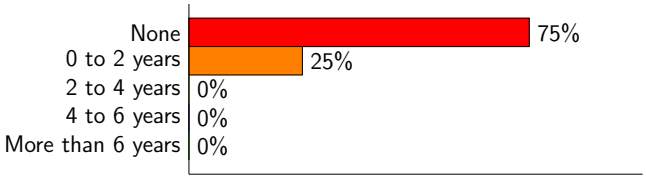
(a) What is your experience with Java?



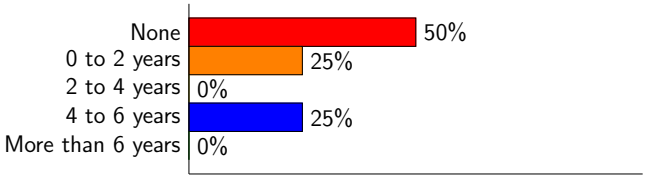
(b) What is your experience with the Eclipse IDE?



(c) What is your experience with the JUnit Framework?



(d) What is your experience with visual debugging techniques?



(e) What is your experience with Java in an industrial environment?

Figure 41 – *Answers 22 to 26*