FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Hooray, I found the bug

João Nadais

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Rui Maranhão

July 22, 2016

Hooray, I found the bug

João Nadais

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Professor Ana Cristina Ramada Paiva
External Examiner: Professor João Paulo de Sousa Ferreira Fernandes
Supervisor: Professor Rui Filipe de Lima Maranhão Abreu
July 22, 2016

Resumo

O processo de verificação de erros tem causado várias dores de cabeça a um grande número de programadores ao longo do tempo. Por vezes, o erro é encontrado numa fase muito tardia do desenvolvimento, levando a elevados custos para a sua correção. Algumas ferramentas dão nos a possibilidade de, a partir de um conjunto de testes, ter uma perceção mais visual do projeto que está a ser analisado, assim como a probabilidade de encontrar o erro em certas partes do código. No entanto, ainda não é possível confirmar o impacto real do uso destas ferramentas no desempenho dos programadores.

Tendo isto em atenção, as atuais ferramentas existentes no mercado com este propósito foram analisadas e uma (Crowbar) foi escolhida para desenvolvimento continuado. Usando uma análise baseada no espetro de código, conseguindo uma maior eficiência do que os restantes métodos, e estando integrado no IDE, o utilizador pode assim utilizar uma nova abordagem para o problema de localização de falhas. Novos tipos de visualizações foram adicionados, bem como opções para filtrar o que é visto para facilitar o foco do utilizador. O impacto destas adições foi avaliado num estudo de utilização que visava comparar não só o facto destas alterações constituirem uma melhoria no desempenho dos utilizadores a localizar falhas, bem como a importância de ter esta ferramenta integrada no IDE e a facilidade de uso da mesma. Este estudo foi feito também por pessoas sem ferramenta, de forma a tirar conclusões sobre a abordagem usada por estes para o mesmo problema.

Do estudo de utilização feito, foram comparados os tempos que cada utilizador demorou, bem como a experiência de cada um. Apesar do possível problema de granularidade nos resultados, foi possível concluir que a experiência do utilizador na resolução de problemas de *software* tem um maior impacto do que o facto deste ter acesso ou não a uma ferramenta visual.

Palavras Chave: *Software*, Testar, Falha, Depuração, Visualização, Interação, Casos de teste

Abstract

Over the years, the process of locating faults in the code has caused various headaches to a large number of programmers.Sometimes, the error is found at such a late stage of development, that correcting it implies high costs. Some tools give us, from a set of tests, the possibility of having a more visual perception of the project that is being analyzed, as well as the probability of finding the error in certain parts of the code. However, so far it is still not possible to state the real impact that having this kind of tool in the general user experience.

Considering this, the available tools in the market with this goal were analyzed and one (Crowbar) was chosen for further development. Being based on algorithms that strive for efficiency, such as Spectrum-based fault localization, and being integrated in the IDE, the user has the possibility of having a new approach to the fault localization problem. New types of visualizations were added, as well as options to filter what is shown to the user, allowing him to focus better. The impact of these additions was evaluated in a user study aimed to compare not only whether these changes improved the user's performance in locating faults, but also the importance of having this tool integrated in the IDE and the ease of use of the tool. This study was also done for people without access to the tool, in order to evaluate the approach used by the developer to find the error.

From this user study, the time elapsed for finding the bug was compared, as well as the users' experience levels in development. Even though the granularity results were a threat to this study, it was possible to conclude that the users' experience locating faults has a bigger impact than the fact of having a visual debugging tool or not.

Keywords: Software, Testing, Bug, Fault localization, Debugging, Visualization, Interaction, Test cases

Acknowledgements

I would like to thank my supervisor Rui Maranhão, for all the help and guidance throughout the preparations and elaboration of the work described in this dissertation.

I would like to thank Luís Cruz for the endless help in making the results of this dissertation as valid as possible.

Also, thank you to Alexandre Perez, whose initial guidance in the first six months of setting things up and gathering knowledge on the topic proved to be useful to elaborate the state of the art.

Thank you to Jorge Costa for giving me insights on how things were done before my development and being always available to help and interested in what I was doing.

Thank you to all the people who participated in my user study. Each and every single one of these people had an undeniable role in what the results of this dissertation turned out.

Thank you to my family for continuous support and helping me to stay focused and on the road, specially when what looked better was quitting.

I could never finish this without mentioning BEST Porto, the local group of Board of European Students of Technology. Thank you to all the members of this group. The past four years (including one and a half managing this local association) have been totally remarkable. Even when I sort of disappointed you and had to resign from my management position you always supported me and understood that what matters the most is that you contribute in what you are best on and what you feel better on. I can't even think of a student life of mine that didn't pass through this organization.

I would also like to thank my management of the Information Technology Committee of BEST of this year, Amato Van Geyt and Dino Memovic. Thank you for helping me covering when I was more busy with this and my availability fell. I couldn't have asked for a better team for sure. This was a memorable year and that's for sure because it was with these people.

For all the support in and outside of BEST, as well as while writing this, I would like to thank Joana Silva. She made me realize that helping does not always mean being on your side. Thank you for all the lessons and for pushing me to try better every time.

João Nadais

"I've failed over and over again in my life...
And that is why I succeed"

Michael Jordan

Contents

1 Introduction **1**
1.0.1 Base project example . . . 1
1.1 Problem Statement . . . 4
1.2 Motivation . . . 5
1.3 Goals . . . 5
1.4 Document structure . . . 6

2 State of the Art **7**
2.1 Fault localization techniques . . . 7
2.1.1 Spectrum based fault localization . . . 7
2.1.2 Spectrum based reasoning . . . 9
2.1.3 Code coverage - Tarantula . . . 9
2.1.4 Program Slicing - xSlice . . . 11
2.1.5 Conclusion . . . 13
2.2 Software visualizations . . . 13
2.2.1 Codecity . . . 13
2.2.2 NDepend . . . 14
2.2.3 Conclusion . . . 14
2.3 Debugging software tools . . . 15
2.3.1 EzUnit . . . 15
2.3.2 DDD . . . 16
2.4 Browser Rendering in Java . . . 16

3 *Crowbar* development contributions **19**
3.1 Visualization features . . . 19
3.2 IntelliJ plugin . . . 20
3.2.1 Implementation of *Crowbar* visualizations natively . . . 20
3.2.2 Rendering browser visualizations . . . 21
3.3 Development in visualizations core . . . 21
3.4 Filtering what is shown in each visualization . . . 23
3.5 Sum up . . . 24

4 User study **25**
4.1 Context . . . 25
4.2 The Project . . . 25
4.3 Problem to solve . . . 26
4.4 Participants . . . 26
4.5 Descriptive Statistics of the sample . . . 27

4.6 Summary 28

5 User study results **29**
5.1 Descriptive statistics 30
5.2 Results comparison 32
5.2.1 No tool vs Full tool 32
5.2.2 Comparison between visualizations 34
5.3 Comparison experience-times 35
5.3.1 Kruskal-Wallis H test 35
5.4 User studies comparison 37
5.5 Sum up 37

6 Conclusions and Future Work **39**
6.1 Contributions 39
6.2 Answer to research questions 39
6.2.1 Does having a visual debugging tool for a user result in better performance locating faults? 39
6.2.2 Does the use of a specific IDE have an impact on their performance? . . . 40
6.2.3 What kind of tools do people use when locating their faults? 40
6.2.4 What helps people find bugs? 40
6.3 Future work 40

References **41**

List of Figures

1.1 First Run of test suite 3
1.2 Amplification in Person class 3
1.3 Result of test cases after correcting the bug 4

2.1 Code coverage with tarantula [JH05] 10
2.2 Example of program Slicing (adapted from [AHLW95]) 11
2.3 Example of program Slicing [Cle02] 12
2.4 Final output with slices highlighted [Cle02] 12
2.5 CodeCity code visualization (from [WL08]) 13
2.6 Dependencies visualization with nDepend on Visual Studio [NDe16] 14
2.7 EzUnit View (from [Phi07]) 15
2.8 EzUnit Call Graph View (from [Phi07]) 16

3.1 Visualization of joda-time project in the two initial perspectives 21
3.2 Tree visualization in joda-time project 22
3.3 Zoomed tree visualization in PeriodType class of joda-time 23
3.4 Effect of filtering visualization on Reingold-Tilford tree of joda-time 24

4.1 Occupation of participants that did not use Crowbar vs participants that used Crowbar 27
4.2 Tools used for debugging from user study participants 28

5.1 Evaluation of user experience by developers that used the tool 31
5.2 Evaluation of usefulness of score-filtering 31
5.3 Times comparison with and without tool 33
5.4 Times comparison within visualizations 34
5.5 Times comparison with different levels of experience in Java 36

LIST OF FIGURES

List of Tables

2.1 Table of program spectra 7
2.2 SFL execution and calculations 9

4.1 Descriptive statistics for Eclipse, Java and JUnit experience 27

5.1 Frequency of bug finding for each case 29
5.2 Descriptive statistics for each of the groups analyzed 30
5.3 Descriptive statistics for questions after user study 30
5.4 Frequency of bug finding for each case 35

LIST OF TABLES

Abbreviations

DDD	Data Display Debugger
GDB	GNU Project Debugger
IDE	Integrated Development Environment
JVM	Java Virtual Machine
MBD	Model Based Diagnosis
MHS	Minimal-Hitting Set
SFL	Spectrum-based Fault Localization

Chapter 1

Introduction

Faults in the code appear in all phases of a software project and need to be solved. The way these faults are dealt with is severely dependent on the developer's knowledge, as well as the tools he/she is familiar with. A The developer can be more used to debugging tools provided by the Integrated Development Environments (from now on referred to as IDEs) such as breakpoints and sequential runnning of the code. However, he/she can also use some other alternatives on the market. These tools may provide a different approach to the same issue, leaving the developer with the decision to choose what fits his/her methods better.

To help on this process, a lot of different approaches and different tools have been used and developed, being *Crowbar*[Cro16] one of them. Having started by the name of *GZoltar* [CRPA12], *Crowbar* is an interactive and visual debugging tool that basing on a test or set of tests generates a visual layer-oriented report showing our code and functions, assigning different colors based on the involvement of each part in failing/passing tests. Some other tools exist within the same concept. However, as it is possible to see in Chapter 2, the core of *Crowbar* provides the most efficient results, being this the main reason why this tool was chosen for development in this dissertation.

For better understanding of how the analyzed tool works as well as its faults, an example is described below as well as its execution.

1.0.1 Base project example

Here is an example of a small software project. Consider a class Person with an ID (identifying number) and a name. The aim is to identify when two people are the same (this happens when they have the same ID). The example code can be seen below:

```
1 public class Person {
2   private int ID;
3   private String name;
4   public Person(int iD,String name) {
5     super();
6     this.ID = iD;
```

```
7       this.name=name;
8     }
9     @Override
10    public boolean equals(Object arg0) {
11      if(arg0 instanceof Person)
12      {
13        Person p = (Person) arg0;
14        return this.ID==p.ID;
15      }
16      else
17        return true;
18    }
19    public int getID() {
20      return ID;
21    }
22    public String getName()
23    {
24      return name;
25    }
26  }
```

To compare two instances of Person, a method called `equals` was created verifying if two instances have the same ID or not. In order to make this check automatic, the following tests were created:

```
1     public void testPersonCreation()
2       {
3         Person p = new Person(0,"A");
4         assertTrue(p.getID()==0);
5       }
6       public void testPersonEqual()
7       {
8         Person p = new Person(0,"Miguel");
9         assertFalse(p.equals(new Object()));
10        assertFalse(p.equals(new Person(1,"Miguel")));
11        assertTrue(p.equals(new Person(0,"Miguel")));
12      }
```

The first test checks a correct creation of an instance of Person. The second one compares different cases of errors that can happen in `equals` method. For instance, there can be another person with the same name who is not the same person (for example there are several people called Miguel, and they are not the same). Taking this code and these test cases into account, an analysis was ran using *Crowbar*. Figure 1.1, shows the resulting report. This visualization divides the whole code in three layers, being the first one (the upper-most of all of them) related to the module itself, the second to the defined classes (Testing and Person class) and finally each function inside each class. The colors of each square are associated with the fault probability in

each. Green squares mean that the probability of the fault being caused by that part is close to 0. The closer it gets to yellow (third square in the bottom layer) the closer it is to 50%, and the closer to red, the closer it gets to 100% probability of being involved in the failure.

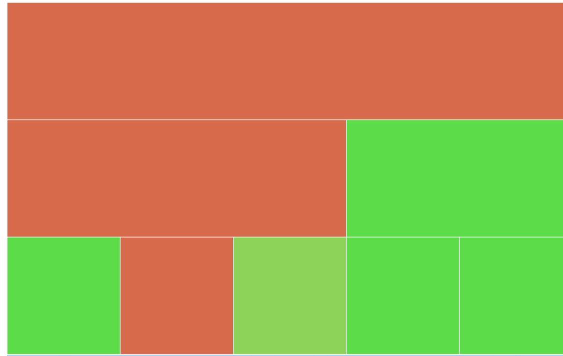

Figure 1.1: First Run of test suite

From this analysis, it is expected that the failure will likely be in the red section of the code (reporting a failed test). To get a better focus on the problem, the second step is to amplify the visualization by focusing on the class that has tests failed. The result of this amplification can be seen in Figure 1.2. There, it becomes clearer that the problem is in Person class and, most likely, in the function in the middle, since the left-most had only successful tests passing there and the other was 50% passed 50% failed. The middle function corresponds to `equals` function.

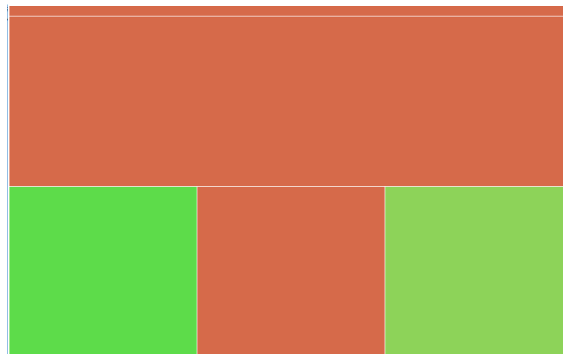

Figure 1.2: Amplification in Person class

Taking a look at line 15 in this function, it is noticeable that there is a `return true` where it should be `return false`, since if two objects are not of the same type they can not be equal. This correction was made and tests were ran again. Analyzing the results obtained by the tests, which can be seen in Figure 1.3, the bug is now corrected.

In this simple example, it was easy to point out the failure as well as to correct it. However, there were some other issues arised.

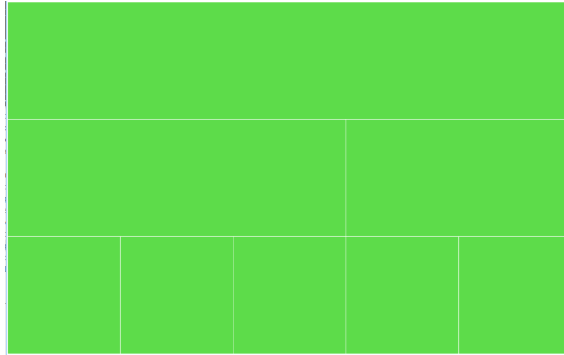

Figure 1.3: Result of test cases after correcting the bug

1.1 Problem Statement

Before an actual product goes to the market, it needs to have a solid base. In this case, *Crowbar* generates the failure probability of each code section by using spectrum-based fault localization combined with spectrum-based reasoning[AZV07]. Although this has proven to be effective with a small example in the previous section, there is still the question on why developing *Crowbar* and why using its algorithm to generate data to help users on debugging.

This dissertation is studying not only the questions about the core of *Crowbar* and its algorithms, but also the improvements that can be done to this tool which were mentioned in the context analysis. One of the problems of this tool is that the visualizations shows all the components instead of focusing only on the parts related to the fault, this leads to unnecessary information in the report, which do not help the user to focus. Also, the code structure is highly hierarchical but the current visualizations do not highlight this hierarchy. Hence, new visualizations need to be investigated, implemented and totally integrated in *Crowbar*.

Finally, it is arguable that *Crowbar* is a good tool for developers' productivity but, without a proper study on it that evaluates the productiveness of users with and without the tool no conclusions can be taken. Therefore, an user study was elaborated on which the testers received an application with a bug, having each developer to find out what the bug was and solve it. By doing this, conclusions could be taken regarding the methods used by developers to locate faults.

This analysis leads into two main focuses for this dissertation:

- Is the current state of the tool enough to help developers locating faults?

- What factors impact the most the performance of developers locating faults?

For the first point, improvements to the tool were defined and implemented, as can be seen in Chapter 3. On the second one, an user study was done and its results as well as its conclusions are reported in Chapter 5

1.2 Motivation

Software developers always struggle with debugging issues, which impact the amount of time used resulting, as well, in a loss of money. Every software project has a budget and a timeline associated, consequently the more time spent in debugging, the less time developers have to develop new features, leading to a more incomplete final project.

Adding to this, by analyzing the strategy of users to locate faults in an unknown project, it is possible to conclude what developers find easier to use for their debugging process, as well as which tools are more commonly used. By combining information on the tools used and the performance of each user, conclusions can be drawn about which tools are more helpful in the debugging process

Finally, with this dissertation it is possible to answer whether or not having a visual debugging tool like the one studied leads to less time spent developing and debugging applications. If that is not the case, it is possible to conclude which are the main factors affecting users' performance when locating faults.

1.3 Goals

For this research project, a set of goals was defined. These goals will allow the improvement of the current state of the tool as well as to increase its value to the productivity of the users' productivity. These goals are:

1. Implement new types of visualizations on the tool and evaluate its usage.
2. Evaluate the impact of reducing the information shown to what is more relevant to the tool users.
3. Evaluate users' performance with and without the tool on debugging a software project.
4. Evaluate the relationship between programming experience and easiness of bug finding.

Adding to these project goals, some research questions should also be pointed out and answered:

1. Does having a visual debugging tool for a user result in better performance locating faults?
2. Does the use of a specific IDE have an impact on the users' performance?
3. What kind of tools do people use when locating their faults?
4. What helps people find bugs?

The answer to these research questions is provided in Chapter 6

1.4 Document structure

This document is divided in four chapters as described below:

2. **State of the art** - Description of the current fault localization techniques, interactive debugging tools, data visualization libraries and libraries for rendering HTML pages in Java applications.

3. **Crowbar development contributions** - Project's development phase. Specification of what was and what was not implemented, as well as the supporting arguments.

4. **User study environment** - Analysis of the problem given to the users, the project and the background of the participants in the user study, namely in terms of experience in fields related to software development

5. **User study results and discussion** - Description of the user study's data, in general and specific. Analysis of the impact in terms of time of using *Crowbar* as well as of having experience in Java. Discussion of the results and conclusions of the study.

6. **Conclusion** - Report of the final results of this dissertation and future work that can be developed.

Chapter 2

State of the Art

2.1 Fault localization techniques

To be able to develop tools that can help programmers to find bugs on their applications, some algorithms and techniques have to be used. Software testers and developers have studied different methods that can help them find bugs more effectively. A survey was done in what regards different types of fault localization techniques [WD09]. Since this dissertation is not only about fault localization but also visual debugging tools, the techniques in this survey were analyzed, presenting here the ones that are used in visual debugging tools.

2.1.1 Spectrum based fault localization

Mnemonic	**Name**	**Description**
BHS	Branch-hit	conditional branches that were executed
BCS	Branch-count	number of lines each conditional branch was executed
PHS	Path-hit	path (intraprocedural, loop-free) that was executed
PCS	Path-count	number of times each path (intraprocedural, loop-free) was executed
CPS	Complete path	Complete path that was executed
DHS	Data-dependence-hit	Definition-use pairs that were executed
DCS	Data-dependence-count	Number of times each definition-use pair was executed
OPS	Output	Output that was produced
ETS	Execution-trace	Execution trace that was produced

Table 2.1: Table of program spectra

Spectrum-based Fault localization (hereafter referred to as SFL)[AZV07] is a fault localization technique based on an analysis of the differences between program spectra. Program spectrum is an execution profile that indicates which parts of the system are active during a run. There are many different types of program spectra, as can be seen in Table 2.1. To be able to perform an analysis through SFL, a failure needs to be detected first, that is, it is needed to know that something is actually wrong before trying to locate the fault and correct it.

In order to understand how this technique works, consider the example in the previous chapter. SFL divides the code in blocks of execution. In the case of the previous example, it is possible to see that it is divided in three execution blocks where a block of code is defined as a Java language statement. For illustrative purposes, the blocks of code from the previous example are defined below:

```
1  @Override
2  public boolean equals(Object arg0) {
3    //Block 1
4    if(arg0 instanceof Person)
5    {
6    //Block 2
7      Person p = (Person) arg0;
8      return this.ID==p.ID;
9    }
10   else
11   {
12   //Block 3
13   return true;
14   }
15 }
```

To illustrate the technique's approach, consider Table 2.2. In this table, each of the tests presented above are considered as inputs. Each of the objects considered is compared with a Person of ID 0 and name "Miguel". A '1' means a hit in the block and a 0 means the block was not affected. Similarity coefficients are also used to calculate the resemblance between each block of code and the faulty code. In this case, two different coefficients are compared. Tarantula coefficient s_T, used in Tarantula, and Ochiai coefficient s_O, used in SFL. The calculation formulas for each of the coefficients are the ones below:

$$s_T = \frac{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)}}{\frac{a_{11}(j)}{a_{11}(j)+a_{01}(j)} + \frac{a_{10}(j)}{a_{10}(j)+a_{00}(j)}}$$

$$s_O = \frac{a_{11}(j)}{\sqrt{(a_{11}(j)+a_{01}(j))*(a_{11}(j)+a_{10}(j))}}$$

In the above formulas, $a_{11}(j)$ means the total amount of times block j was executed with all the inputs and returned an error. Similarly, $a_{10}(j)$ gathers the total amount of times it was executed and the output was successful, $a_{01}(j)$ is similar for not executed and error, and finally $a_{00}(j)$ for not executed and successful. Taking these coefficients into consideration, three inputs are considered for this example (I_1,I_2 and I_3). I_1 represents a general object. Since it is not even an instance of Person, it should return false. I_2 and I_3 are both instances of person, so none of them is involved in block 3. Table 2.2 demonstrates the calculation of the similarity coefficients for these inputs.

By analyzing the values from the similarity coefficients, it is possible to conclude that the error will be in block 3, since it is the block with the biggest values in both similarity coefficients.

Input	Block 1	Block 2	Block 3	Error
I_1 = (new Object())	1	0	1	1
I_2 = (new Person(1,"A"))	1	1	0	0
I_3 = (new Person(0,"A"))	1	1	0	0
s_T	0.5	0	1	
s_O	0.577	0	1	

Table 2.2: SFL execution and calculations

2.1.2 Spectrum based reasoning

SFL performs worse when a failure is caused by various faults with no specific correlation between them [AZvG07]. This makes it necessary to combine this technique with some other that addresses the aforementioned failures. Spectrum-based reasoning results from the combination of SFL with Model Based Diagnosis (MBD), which allows SFL to deduce multiple-fault candidates, while using heuristics to avoid MBD's exponential complexity to a point where it can be used on large, real-world programs without any problems.

In the Spectrum-based reasoning approach, each component is modeled in terms of the following logical proposition[AZvG07]

$$h_j \implies (ok_{inpj} \implies ok_{outj})$$

In the proposition above, h_j, ok_{inpj} and ok_{outj} stand for the model component health and the (value) correctness of component's input and output, respectively. According to this formula, a failure can be noticed if the correct inputs are used but the output is not as expected. However, it is still possible that the input is incorrect and still giving correct results, which proves to say that there might be a fault, even though there is a correct output.

The health propositions are then combined using a MHS algorithm. This is usually a part that takes a significant amount of resources in MBD approaches. Therefore, it is used an heuristic algorithm called STACCATO (STAtistiCs-direCted minimAl hiTing set algOrithm) that is based on a heuristics which uses a very low amount of resources, keeping the algorithm usable.

All in all, this algorithm aims for the combination of the missing points of SFL, the probabilistic and multiple-fault localization possible with MBDs, then reducing its resource wasting my cutting the search to the very specifically needed and using STACCATO as a low-resource algorithm to combine health propositions, while still being able to perform at a decent level in different types of machines.

2.1.3 Code coverage - Tarantula

Tarantula [JH05] is a technique that analyzes code coverage of each test case, assigning colors to each statement of the code according to its likelihood of being faulty, based on the assumption

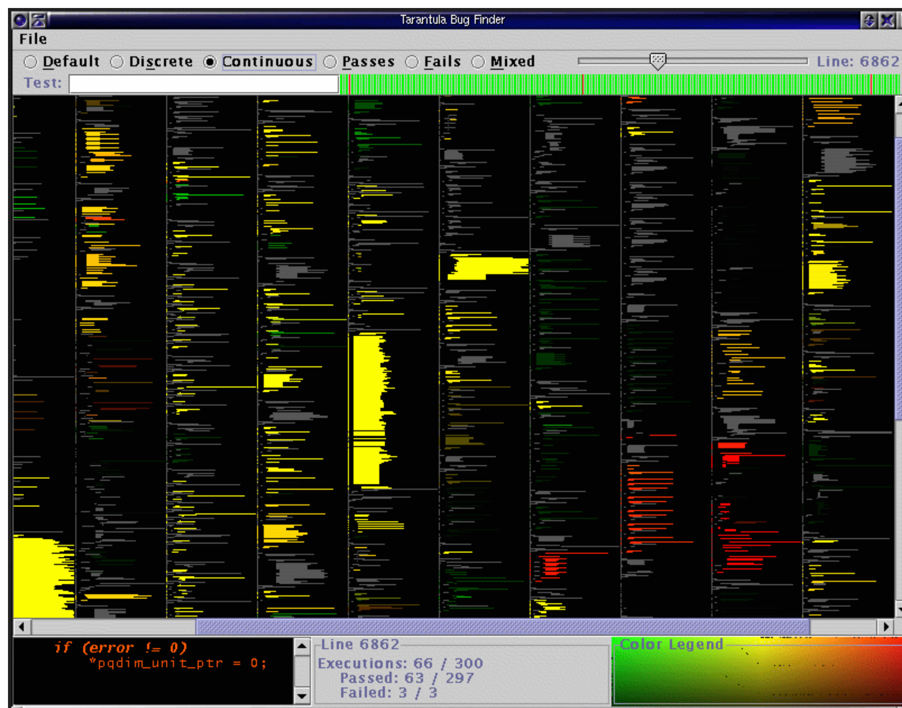

Figure 2.1: Code coverage with tarantula [JH05]

that statements executed primarily in failed test cases are more likely to be faulty than in accepted cases. The most suspicious statements are colored in red, statements ran in only accepted cases are colored green and those who are being executed in both failed and accepted cases are colored yellow (to work as warnings). In particular, the hue of each statement **s** is calculated according to the the number of tests passed that execute **s** (represented as Passed(s)) and the number of tests failed that execute **s** (failed(s)). The equation [JH05] is shown below:

$$hue(s) = \frac{\frac{passed(s)}{total\,passed}}{\frac{passed(s)}{total\,passed} + \frac{failed(s)}{total\,failed}}$$

This hue value also gets to the purpose of evaluating how likely it is for the fault to be in **s**, working on reverse of this value (those with the higher value are less suspicious and keep increasing until those with smallest value). This value can then be used as a ranking to analyze statements, on which the higher values have higher rankings and thus are the primary choice for searching the bug. A study conducted by Jones and Harrold [JH05] revealed Tarantula required the examination of less than 10% of the application code in order to detect the fault.

Figure 2.1 shows an example of an execution of Tarantula. In that example, 300 tests are run and 3 are failing. These 3 tests are analyzed, calculating a fault probability. This fault probability is responsible for the color gradient observed in Figure 2.1, from green for section not involved in the failure to red for sections in which the failure is more likely.

2.1.4 Program Slicing - xSlice

xSlice[Cle02] is based on Program slicing technique[AHLW95], which has two main concepts: slices and dices.

There are two types of slices: static slices and dynamic slices. A static slice is the set of statements that might affect the value of a particular output, while dynamic slices are those which affect output value, given a specific input.

A dice is the difference between two slices. These dices are used to filter a bigger slice into smaller ones. For example, when a slice contains a fault, there might be a set of lines which are not involved in that fault. To evaluate the result, the obtained dice is composed by the original slice with the not faulty lines taken off. A more visual example of how this technique works is observed in Figure 2.2. In this example there are two different slices involved in bugs, called A and B. Both of them are involved in bug A, so the dice where this bug happens will be the one with the lines that run in both slices. On the other hand, bug b only happens in slice B (not in slice A) so, the resulting dice will be all the lines of B that are not run in A (Dice A-B, in red in the figure)

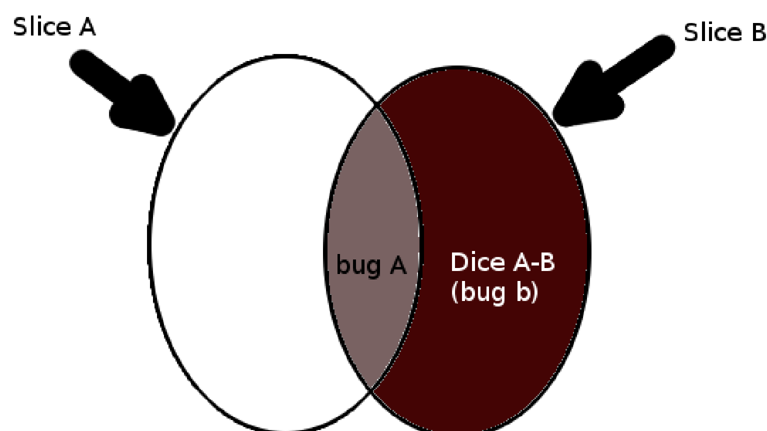

Figure 2.2: Example of program Slicing (adapted from [AHLW95])

xSlice is an application based on program slicing technique. It considers a set of tests failing and divides each file into slices according to whether they are or not involved in the fault. Figure 2.3 shows the initial output of running an application's tests in xSlice. It is an application that counts the number of lines, words, and/or characters on its input.

Three tests were ran on this file, using as input the following line (the first character is a tab):

test input file 1

This input expects that the program counts 4 words, 19 characters and only 1 line. Test 1 checks all these parameters and is successful both in the faulty and working application. Test

Figure 2.3: Example of program Slicing [Cle02]

2 checks words only and the result is 4 in both. However, Test 3 generates an empty line in the faulty application, and the value 4 (counting words only) in the working application. Using program slicing technique produces the output observable in Figure 2.4. Each slice of code is assigned a different color depending on whether it was run in each of the test cases. As can be seen by the figure, there are 4 lines in the file that were involved in the failing test case and were not involved in the others. This gives the developer a good starting point to find the fault. In this case, the problem was that it was processing character by character instead of line by line, making it return an empty string when it should not do it.

Figure 2.4: Final output with slices highlighted [Cle02]

2.1.5 Conclusion

In this section multiple methods and technologies for fault localization are observed. However, when comparing the efficiency of *Crowbar*/SFL with others such as Tarantula, the similarity coefficient used by SFL outperforms the one from Tarantula, making *Crowbar* a better option. On the other hand, although both problem slicing and SFL use extracts of code to help identify faults, there is no direct comparison for the same problem using *Crowbar* or xSlice making it impossible to conclude which of them has a better performance, nor there are indications on performance issues on xSlice. Hence, since *Crowbar* is proven to work well and be effective with big projects, while the same is not proven with xSlice, the choice goes for *Crowbar*.

2.2 Software visualizations

2.2.1 Codecity

CodeCity [WL08] is an integrated environment for software analysis, displaying the project code as an interactive 3D city. The packages are represented by districts in the city, and each of the buildings represents a class in a package. Each building's height is defined by the number of methods of the corresponding class. The more methods the class has, the higher the building is. Similarly, the width varies in the same proportion as the number of attributes of the class. Figure 2.5 provides an example of a code sample represented in CodeCity, in this case ArgoUML, a Java system for handling UML diagrams.

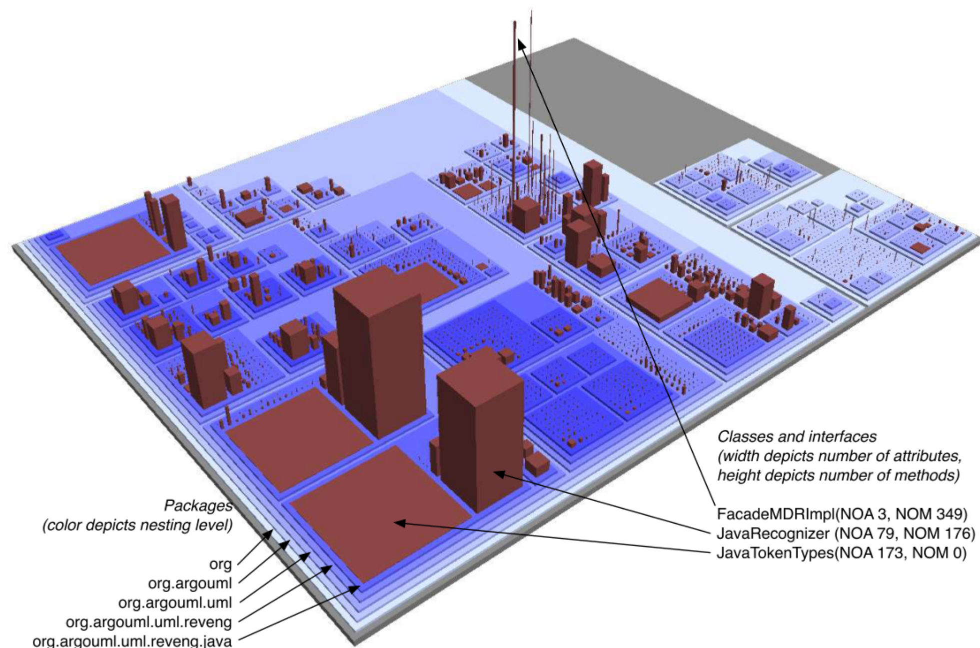

Figure 2.5: CodeCity code visualization (from [WL08])

This visualization style had its first implementation with OpenGL, but due to its success, an *Eclipse* plugin was also created for it, as well as a new version using WebGL. Consequently, these types of visualizations can now be observed directly with a browser.

2.2.2 NDepend

NDepend [NDe16] is a Visual Studio plugin of static analysis for .NET code. It has two main types of visualizations: the first one (graph-based) generates a graph from the code and links different sections in case they are dependent on one another. The second one evaluates code coverage, representing each method as a rectangle and having a gradient color according to the tests' coverage of the coverage of the method. 0% coverage results on coloring that rectangle as red, and as the coverage increases the color changes gradually to yellow and finally to green where there is total coverage. An example of this procedure is observed in Figure 2.6.

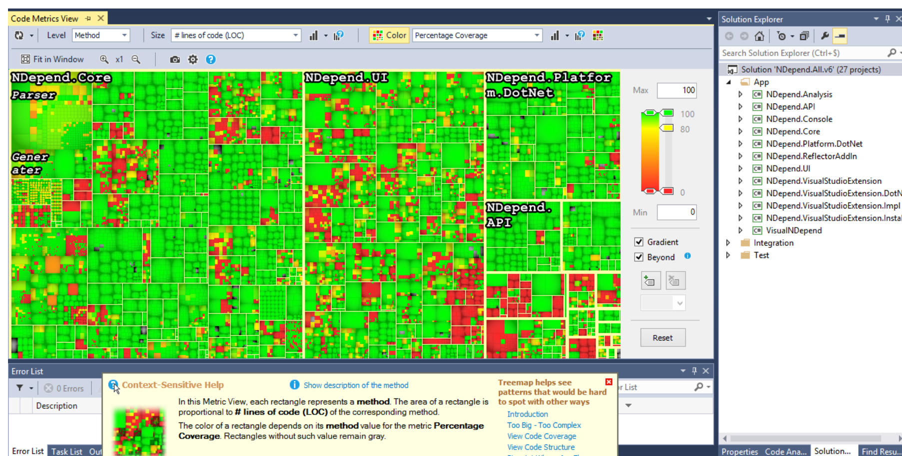

Figure 2.6: Dependencies visualization with nDepend on Visual Studio [NDe16]

2.2.3 Conclusion

These software visualizations could be an alternative to the visualizations provided by *Crowbar*, which uses data from SFL and HTML5 libraries. However, using any of these two examples in an IDE would require a correct scale of the different ranges of windows as well as have interaction with what is shown to the code related. In the case of Codecity, the visualizations are generated using WebGL, and the current implementation for IDEs renders the visualization in a browser instead of running it natively, implying that the interaction with the IDE would be lost. NDepend uses similar concepts as the ones used by *Crowbar*, such as gradients for code coverage of tests. In the case of *Crowbar*, gradient is used according to the probability of the fault being in a specific part. Since NDepend and *Crowbar* have a similar concept but work differently, we can only use *Crowbar* to help on debugging process, being able to use NDepend for test coverage checking.

2.3 Debugging software tools

The market of debugging tools has a wide variety of solutions and approaches to the same goal. In the user study that happened, the users that did not have *Crowbar* could use any other tool to approach the problem. Hence, a study on the current existing tools was done to see what was available to them. The following subsections will explain in more detail some of the biggest tools in the market and how they can be compared with *Crowbar*.

2.3.1 EzUnit

EzUnit[Phi07] is a plugin for *Eclipse* that connects test failures to the source code of the application. Given a Java project with JUnit tests that fail, we execute the application, and by opening EzUnit view in *Eclipse* we can get a similar scenery to the one appearing in Figure 2.7. This figure shows an example of an application to manage money in bags, pointing for an issue in the addition of more money.

EzUnit Result

type filter text

Avg	Method Under Test	Resource	Path	Location
0.85	Money.addMoney(Money)	Money.java	/FullMoney/junit/samples/money/Money.java	25
0.85	Money.add(IMoney)	Money.java	/FullMoney/junit/samples/money/Money.java	22
0.83	Money.toString()	Money.java	/FullMoney/junit/samples/money/Money.java	70
0.73	MoneyBag.appendTo(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	121
0.73	MoneyBag.appendBag(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	35
0.69	Money.equals(Object)	Money.java	/FullMoney/junit/samples/money/Money.java	40
0.64	MoneyBag.toString()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	113
0.55	Money.<init>(int,String)	Money.java	/FullMoney/junit/samples/money/Money.java	-1
0.55	MoneyBag.add(IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	26
0.52	Money.currency()	Money.java	/FullMoney/junit/samples/money/Money.java	36
0.52	Money.isZero()	Money.java	/FullMoney/junit/samples/money/Money.java	57
0.50	Money.amount()	Money.java	/FullMoney/junit/samples/money/Money.java	33
0.49	MoneyBag.findMoney(String)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	70
0.49	MoneyBag.appendMoney(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	39
0.49	MoneyBag.create(IMoney,IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	20
0.49	MoneyBag.<init>()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	-1
0.49	MoneyBag.simplify()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	104
0.42	Money.appendTo(MoneyBag)	Money.java	/FullMoney/junit/samples/money/Money.java	75
0.41	Money.negate()	Money.java	/FullMoney/junit/samples/money/Money.java	63
0.36	MoneyBag.subtract(IMoney)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	109
0.36	MoneyBag.addMoney(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	29
0.36	MoneyBag.addMoneyBag(MoneyBag)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	32
0.29	MoneyBag.negate()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	98
0.23	MoneyBag.isZero()	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	88
0.18	Money.subtract(IMoney)	Money.java	/FullMoney/junit/samples/money/Money.java	66
0.18	Money.addMoneyBag(MoneyBag)	Money.java	/FullMoney/junit/samples/money/Money.java	30
0.16	MoneyBag.contains(Money)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	76
0.16	MoneyBag.equals(Object)	MoneyBag.java	/FullMoney/junit/samples/money/MoneyBag.java	53
	Fault Circle (S BSA BS BSA SA SS N2 IZ N4 SBA N3)			

Figure 2.7: EzUnit View (from [Phi07])

As noticed before, sometimes faults are located in functions which are called multiple times in different places of the application. To counteract this, EzUnit gives the possibility of generating a call graph for each test case, considering again the previous color mapping, as observed in Figure 2.8. In this example, the focus is on the first test case from Figure 2.7 which was the main suspect of where the bug might be. By analyzing this call graph the color differences are noticeable, being easy to spot the function that is more likely to have caused the error.

2.3.1.1 Comparison

EzUnit is similar to *Crowbar* as both tools are focused on visual and interactive debugging based on test cases. However, EzUnit is only limited to *Eclipse* IDE, whereas *Crowbar* can run in *Eclipse* but also independently, as a Maven plugin. Finally, EzUnit only allows visualization of the code as a table and a call graph, while *Crowbar* provides both the table and visualizations of the code structure that were shown in Chapter 1.This type of visualizations do not have a parallel in EzUnit.

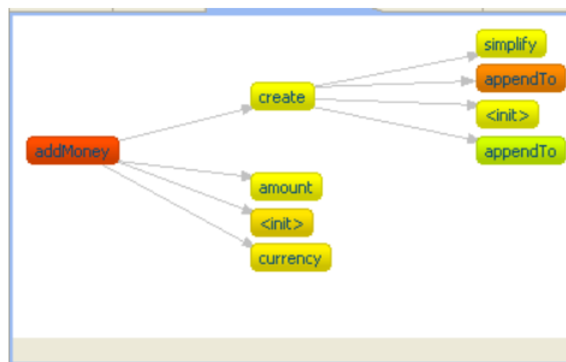

Figure 2.8: EzUnit Call Graph View (from [Phi07])

2.3.2 DDD

DDD [ZL96] is a graphical front-end for command-line debuggers such as Pydb, GDB and bashdb. Before starting DDD, it is necessary to compile the code to be tested using a command-line compiler. After that, DDD is executed with the executable file resultant from the compilation. By doing this, DDD opens and it is possible to analyze the code, setting up break points in case the user wants to re-run it step by step. While debugging, if data structures are present, they are shown in a graph-mode, as well as the data they store. If the user wants to go more into detail, DDD offers the possibility of analyzing the records being changed, their next instructions and current values.

2.3.2.1 Comparison

Despite the fact that both tools work as visual debuggers, there are some differences that can be pointed out when comparing the two:

1. **Number of languages supported** - DDD does not have any integrated debugger so it supports a wide variety of languages provided that the user has a shell compiler for it. On the other hand, *Crowbar* is only supporting Java

2. **Code coverage** - this is only present on *Crowbar* as it is based on tests and analyses their coverage, while DDD focuses on the most common sense of debugging and step by step execution

2.4 Browser Rendering in Java

In order to support visualizations based on D3 and javascript in Java based IDEs (the visualizations shown in Section 1.0.1) and to have an IDE plugin for *Crowbar*, it is necessary to have a way to render HTML-based visualizations in Java applications. For this purpose, and since Java version 8 on which JavaFX was launched, there is native support to render HTML pages in Java applications using JavaFX. However, IDE plugins can not be made yet with JavaFX (only with Swing) leaving this possibility as not implementable.

However, by itself *Eclipse* already supports a native browser, being then possible to render webpages inside the IDE. For this, it uses its Standard Widget Toolkit (SWT)[The16]. SWT takes the most of the libraries used for this native browser to render HTML pages in Java applications. Moreover, if newer features from HTML are needed (for example HTML5 features) it is possible to customize SWT Browser library to render the page using an installed library in the computer (for example webkit library), allowing better flexibility as well as better rendering of the results. All code visualization figures observed in Chapter 3 had as a base Eclipse SWT with native webkit library.

If it is not desired to be dependent on that IDE (or wanting to implement in other IDE) there is still some other possibility, for example JxBrowser[Tea16]. This external library allows the developers to render a HTML page in a Java application (or, in this specific case, a Java plugin), whether it is based on Swing or JavaFX. The integration of this library allows to users the same possibility that they had by using *Eclipse* SWT.

Chapter 3

Crowbar development contributions

Crowbar is composed of several modules. One of these modules (reports generation and visualization) was the focus of this dissertation. For this part, the main question was: how can all the information generated in the core be shown such that users can understand it and interact with it?

One of the most important points to consider is that developers usually take advantage of the features of the IDE for their debugging processes. For example, the use of the IDE debugger and breakpoints in IDEs like *Eclipse* allows the user to run each test step by step. According to a user study [MKF06], debugging views in IDEs are widely used by developers, being noted as most used IDE debugging features the possibility of inspecting variable values, checking information on running threads and stacks, and finally execution control. Taking this into account, and since *Crowbar* also aims to be a debugging tool, the focus of developments proposed in this dissertation and presented in the user study was on IDE plugins that can integrate *Crowbar* features with the ones from the IDE.

Having taken this into consideration, a research on the most-used Java IDEs was analyzed. Even though up to 2005 it was safe to say that *Eclipse* was the most dominant IDE in the market[Got05][Gee05], at the time, IntelliJ IDEA[s.r16] was still starting, and its recent growth in popularity raises the question on if it is still the case or not. Hence, development was then divided in two parts: on the one hand, it was searched upon the possibility of implementing a plugin for IntelliJ IDEA, how it would work out and interact with the IDE, having a common ground and core with *Eclipse* IDE plugin already implemented. On the other hand, another aim is to improve the core of the plugin (visualizations module of *Crowbar*), adding new types of visualizations as well as new components for processing each visualization.

3.1 Visualization features

The visualizations module is composed of several sub-modules, also called components, which specify the interactive features from the application. Some examples of these are:

- **Zoom controller** - allows zooming in a specific part of the visualization. For example, if the visualization is focused on the package, it is possible to select a sub-package/class inside

it and zoom in it. By zooming in, only the parts directly related to the package can be seen (Figure 3.3 is an example).

- **State manager** - allows the user to go back to the previous zoomed section or redoing a zoom that was undone by using the left and right arrows in the top-left corner of the screen.
- **Key Bindings** - allows the user to change by pressing specific keys in the keyboard, instead of clicking on the visualizations. For example, by pressing key 2, it will automatically change to the visualization in the second tab.
- **Node info** - when hovering a node or a section in the visualization, it is possible to see in the upper-left corner the path to reach that specific node. This allows the user to associate one part of the visualization to a part of the project he/she is debugging.
- **Section tooltips** - in parallel to what happens in Node info, when hovering a node a tooltip appears in the node hovered describing if this node is involved or not in passing/failing tests and if yes, in how many of each.

These visualization features are, as of now, fully implemented in all the visualizations from the tool. However, during this dissertation, the possibility of filtering a visualization by its score was also added. More details are provided in Section 3.4.

3.2 IntelliJ plugin

Android Studio, the Jetbrains IDE for Android which is based on IntelliJ IDEA, has a module called Allocation Tracker that analyzes memory allocation while an application is running. This data is logged and visualized using native Java visualizations that look similar to sunburst visualization available in *Crowbar*. By taking the most out of Android Studio's community version, being able to access the source code on how visualizations are generated, there were two possible ways that could be followed to get *Crowbar* in IntelliJ: either re-implementing the whole visualizations natively (based on the source described) as well as the whole logic associated with them that is already done in *Eclipse*, or finding a way to make IntelliJ render browser visualizations. Next sections explore each alternative separately and present the respective conclusions.

3.2.1 Implementation of *Crowbar* visualizations natively

Analyzing what Android Studio had to offer with Allocation Tracker, as well as the source code that generated the sunburst visualization, provided a starting point on how to go through with this plugin for IntelliJ. However, implementing the plugin to work with native visualizations would mean rewriting the whole logic of the plugin and its interaction with the *IDE*, as well as the visualization rendering and data transferring from *Crowbar* core data processing to the visualization module. Since this would be a bigger effort than just a simple porting from Eclipse to IntelliJ, considering a future maintenance and as *Crowbar*'s Maven plugin also uses HTML-based visualizations, the option chosen was to find a way to render browser visualizations.

3.2.2 Rendering browser visualizations

Rendering browser visualizations within Java applications was the main priority of implementing *Crowbar* in IntelliJ. However, IntelliJ does not have any kind of browser inside the IDE, as opposed to what happens in *Eclipse*. Therefore, it required a search for possible implementations of native browser visualizations. After gathering information from IntelliJ development community, it was concluded that there were only two options left: implementing the plugin using JavaFX (that supports Browser Views and rendering) or getting an external library to work on this part. Unfortunately, IntelliJ plugins can only be implemented using Swing, so the first option was rejected. One of the libraries suggested to proceed with this implementation was JxBrowser. Considering the dimension of the library, the amount of goals aimed for this dissertation and the schedule planned, it was decided that the implementation of this part would be given lower priority, and relegated for future work.

3.3 Development in visualizations core

In order to improve the current types of visualizations and implement new ones, the visualization core module was analyzed. Firstly, it was stated that the only possible ways of visualizing information and code structure was through two visualizations: vertical partition (the one shown in the introduction) and sunburst. Both of them are generated based on information from Crowbar core, as well as the fault probability of each part. Figure 3.1 shows the look of the project used in the user-study by the only two available visualizations before this dissertation.

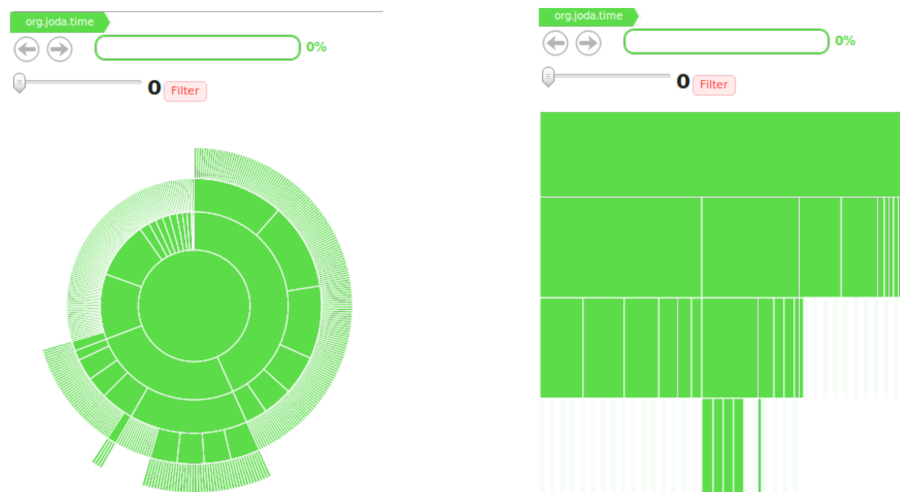

Figure 3.1: Visualization of joda-time project in the two initial perspectives

Secondly, by recognizing that both visualizations are layer-based, while the projects developed are, in fact, a hierarchy, which are commonly represented by a tree, a tree visualization was proposed. In a tree, the root nodes are the most generic ones (in this case it is the root package), having different edges along each sub-package, ending in a leaf node for each of the methods. This type

of visualization was analyzed. From this analysis, several implementations and examples were found that could fit this case. However, the typical approach for tree-based visualizations would have scaling problems on bigger projects. For example, it would be difficult to visualize if there were multiple classes and sub-packages, each of them with multiple functions. Nevertheless, it is possible to scale the tree correctly by taking one of two approaches: a top-down approach (similar to vertical partition's approach), or a circle-based approach, similar to what is observed in sunburst for example. The most interesting of these two is is the circular approach which produces the so-called circular trees. Using Reingold-Tilford algorithm [RT81][SR83] it is possible to implement it in the most space-efficient way as it focuses in space optimization by minimizing the space between nodes in the same level, avoiding overlapping edges. The algorithm works according to the following steps:

- Calculating a minimal distance within the siblings and apply it in each level of the trees
- Shift the overlapping subtrees to the right, with the same distance.

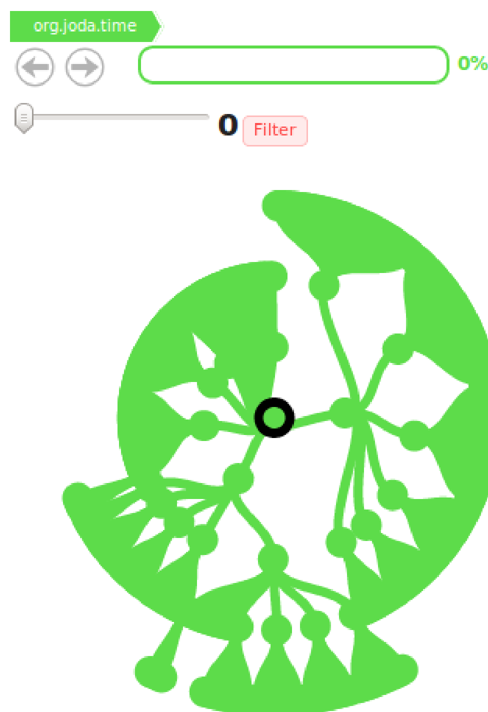

Figure 3.2: Tree visualization in joda-time project

By following these steps, the space is rationalized, a necessary feature as *Crowbar* visualizations aim for good scaling in a wide range of screens. Figure 3.2 describes the same project as the two previous visualizations, but using a circular tree based on Reingold-Tilford algorithm. By clicking on one of the nodes the whole tree gets re-drawn with the node clicked at the center, as observed in Figure 3.3 for the case of PeriodType class in joda-time project. In this example it is possible to

observe the root node (root package) with the circular black border, and different edges connecting all the nodes in the circle. In compliance with Reingold-Tilford property that all nodes in the same level should be at the same distance, the nodes are all connected to the PeriodType node in the center (inside the green spot).

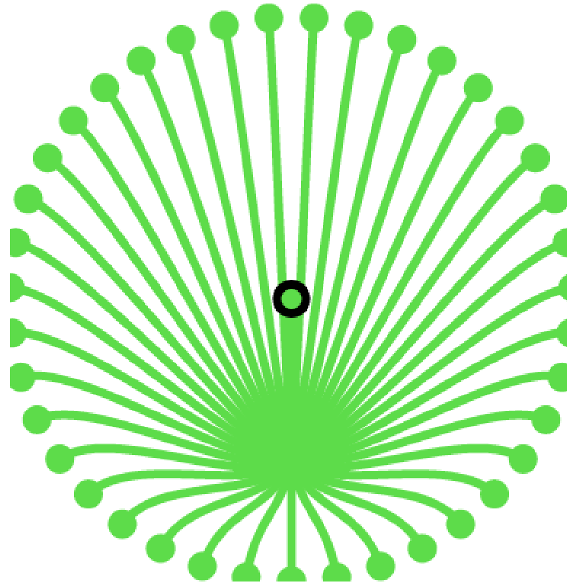

Figure 3.3: Zoomed tree visualization in PeriodType class of joda-time

3.4 Filtering what is shown in each visualization

Each report is based on two main concepts: visualizing the data and assigning a color gradient according to the node's fault probability. When it is considered a small project with 1 or 2 packages of 8-10 classes each, the resulting report will also be small, such that space optimization and filtering the accessory information is not so relevant. On the other hand, projects of bigger scale such as joda-time (the project used in the user study) result in big reports, which are more difficult to visualize in the small screen where visualizations are shown, leading to scaling issues. If it was possible to filter the visualization by focusing only on the part that is most likely related to the fault, these issues would be reduced.

Figure 3.4 shows an example of applying filtering to joda-time project. On the left side is the original tree-based visualization with all its different packages and classes and the filter is set at 0 (top-left corner), so all the branches of the visualization are shown, making navigation more difficult. On the right side is the tree resulting from applying a filter such that only the nodes with 21% of fault probability or higher are shown, i.e., only those that are involved in failing tests with a probability of 21% or more. Consequently, the amount of information shown is greatly reduced, allowing the developer to focus only on the parts of the code that are related to the fault, and the

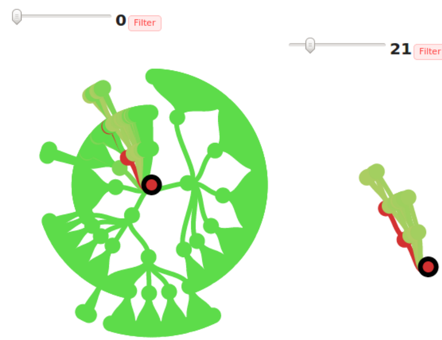

Figure 3.4: Effect of filtering visualization on Reingold-Tilford tree of joda-time

interaction with the visualization gets easier. The impact of implementing this filter possibility on users debugging activity was evaluated in Chapter 5.

3.5 Sum up

The implementation part of this dissertation focused mainly on the core of visualizations of *Crowbar*, as this would be the part of the tools that the users doing the user study would mostly interact with. Having taken into account the current visualizations and its associated features, a new type of visualization (Reingold-Tilford tree) was added to *Crowbar* to allow for a more hierarchy-based view while facilitating the tool's use. Besides this, a new component was added allowing the visualization to be filtered by its score. By filtering the results, less points have to be considered, making it easier to focus on the parts of the code that were mostly affected by the fault.

In what regards the IntelliJ part, as was explained before, it was investigated but not developed much further due to time constraints, ensuring that the user study and feature development in the *Eclipse* plugin of *Crowbar* would be done until the end. The analysis on how to proceed in IntelliJ was already studied and, consequently, can be a point of continuous development on this tool, but left now for future work.

Chapter 4

User study

4.1 Context

As mentioned in Chapter 1, visualizations of coding projects might seem to help users to find faults in their projects and, consequently, develop faster. However, until now, there is no proof on whether this is true, if there is a real impact in users' performance or if and how such tool can help. To evaluate this fact a group of people was gathered to perform a user study. The focus was on people with experience in Java and developing in general, who could have a pure IT background (such as studying Computer Science or Computer Engineering) or just experience in developing from side projects. The education level of these people varied from students proceeding a Bachelor or Master's degree to those that finished their education and are now full-time workers and the university and/or country where their education took place was also different. Nevertheless, all of them were exposed to the same project with the same error and the results of their performance with and without tool were recorded.

4.2 The Project

Previous studies in software testing research have already been done. However, they all faced the same issue: there is a lack of databases of reproducible bugs. For this purpose, the database Defects4J [JJE14] was created. This database provides real reproducible bugs in 5 different open source projects in Java. At the moment, the projects considered in this database are:

- JFreeChart - a Java Chart Library
- Closure Compiler - Javascript compiler and optimizer by Google
- Apache commons-lang - Helper utilities for Java.lang API, made by Apache
- Apache commons-math - Mathematics and statistics library by Apache
- Joda-time - standard Java time library (up to Java 8)

The current implementation of *Crowbar* can only consider Maven-based projects. From these 5 projects, two of them (JFreeChart and Closure Compiler) were not Maven-based, and, therefore, had to be excluded from our possibilities. For the remaining 3, the type of problems reported as faults in each of them, as well as its complexity was analyzed. Since *Crowbar* is based on Maven, and all Maven projects have some modules that are started by `org.apache.common`, these modules are ignored in the core, in order to process only modules related to the project itself and not to generic Maven configurations. This implies that core modules are ignored in both lang and math project. As one of the goals of the user study is to evaluate the usefulness of a tool as compatible as possible with the project to use, these two projects were also discarded leading to the selection of joda-time.

4.3 Problem to solve

Joda-time library has a wide variety of modules that can be evaluated. For this study, the goal was to use a module where it would be possible to isolate and solve the fault within an hour. After analyzing the different modules of Joda-time, Period constructor was identified as the most isolated place on which a fault could be located. This module specifies a period of time in all time quantities (from years up to milliseconds). Each period has a type associated, which defines the quantities that should be processed and the corresponding values. For example, a Period of type DayTime should only consider values associated with days and time of the day, leading to an error every time months or years are observed.

In test-driven development, a bug is detected when a test fails. Hence, since *Crowbar* is based on tests and its success/failure, mirroring a situation of real bug finding would mean facing a test/set of tests that is failing and correcting it. To produce these failing tests in the Period constructor module, some parameters were misplaced and others were input with incorrect values. For the participants to complete the user study, they would have to run the code, analyze the tests, check what was failing and correct this failure. Only when all tests returned as "passed" their user study would be considered as completed successfully.

4.4 Participants

In total, 60 people were contacted to participate in this user study, from which 42 managed to participate and contribute with data. As mentioned in Section 4.1, the participants knew Java and programming, be it either from their studies or from outside projects. In order to assess the experience of each of the participants, the following questions were asked first:

1. Occupation

2. Years of experience with Eclipse

3. Experience in Java Development

4. Experience with JUnit

5. Tools used for debugging software

4.5 Descriptive Statistics of the sample

Since this user study considers two cases (with and without tool), the statistical analysis of the sample divides participants into these two categories.

In terms of occupation, as summarized in Figure 4.1, all of the group that used the tool was composed of master's students, which were also the majority on the group without tool. However, in this group there were also some bachelor students as well as some people that are already working either as developer or as an analyst. The values for mean and standard deviation regarding the technologies used are reported in Table 4.1.

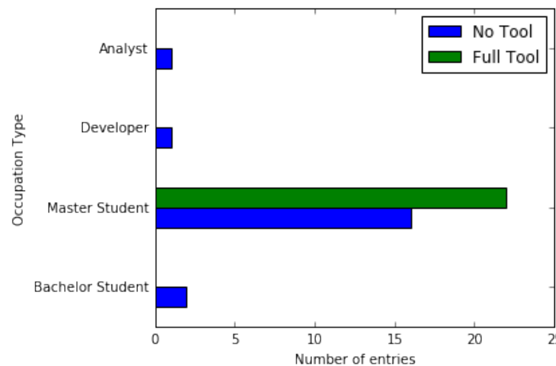

Figure 4.1: Occupation of participants that did not use Crowbar vs participants that used Crowbar

Parameter	Mean	Standard Deviation	Minimum	Q1	Q2	Q3	Maximum
Eclipse experience	2.667	1.087	0.5	1.5	2.5	3.5	4.5
Java experience	2.786	0.995	0.5	2.5	2.5	3.5	4.5
JUnit experience	1.357	0.783	0.5	0.5	1.5	1.5	3.5

Table 4.1: Descriptive statistics for Eclipse, Java and JUnit experience

From Table 4.1 above it is clear that more than 75% of the participants had at least one year of experience with eclipse and more than 2 years of experience with Java, the programming language used in the study. However, only less than 25% of the participants (Q3) had more than 2 years of experience with JUnit and none had more than 4 years.

Also interesting and important for this experience was to know what kind of tools are most commonly used when debugging. Figure 4.2 shows that most participants use prints in the code (40 out of 42) and breakpoints in the IDE (38 out of 42), while only a small sample of the participants use the gnome debugger (4 out of 42).

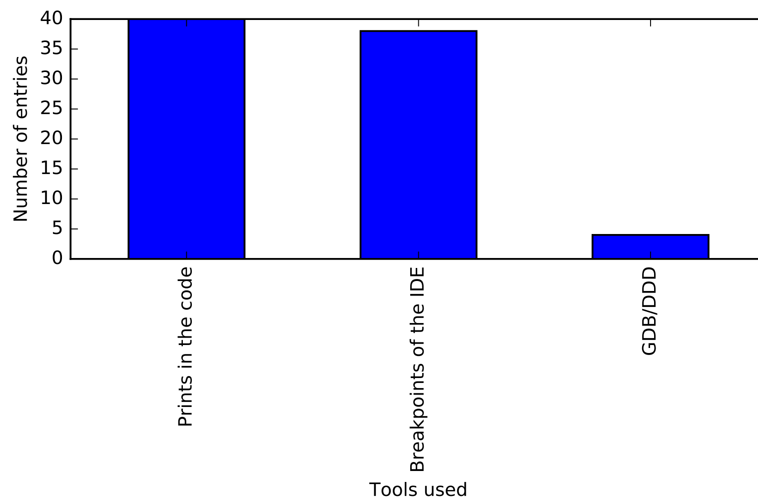

Figure 4.2: Tools used for debugging from user study participants

4.6 Summary

In total, there were 42 participants in the user study. Even though most of the sample had the same education level(last year of college/4th year of college out of 5), the charts and tables in the sections above report that they have different levels of experience with the tools and programming languages. This outcome may be explained by the fact that even though the bases from college are the same, each person deepens his/her knowledge in some specific parts that suits their goals better and, consequently, the languages that they are more experienced with are of a wider variety than the ones taught at the university.

In what regards the project and the user study itself, it was taken into account the amount of time that participants would need to complete it, including the time needed for participants to understand what the project's code was supposed to do, in order to be able to locate the fault and correct it. Comparison between the two groups in terms of time are described in Chapter 5.

Chapter 5

User study results

Approximately 60 developers were contacted to participate in this user study. However, due to different availabilities as well as restrictions in their locations, only 42 managed to participate and submit data that can be considered for this study. As an initial assumption for results, and considering the conclusions of the user study from Gouveia [Gou13], where the difference between having a tool and not having a tool was approximately 20 minutes, the final times of this user study were split among 4 intervals of 20 minutes each, which were then assigned a value from 1 to 4 providing a sequential mapping of the results. This way, number 1 was assigned if the person spent up to 20 minutes, 2 for 20-40 minutes, 3 to more than 40 minutes and 4 for those who did not find the fault. A summary of the number of people in each interval is observable in Table 5.2 where, as done before, the population of the study is divided in two groups (with and without tool) and the results reported separately. On its hand, the group with access to the tool was then divided according to the visualization they used the most. Table 5.2 describes the sample for each of the groups, as well as the mean and standard deviation of the final time intervals defined before.

Time spent	**Without tool**	**With tool**
0-20 minutes	9	13
20-40 minutes	5	4
40+ minutes	2	3
Didn’t find it	4	2
Total	**20**	**22**

Table 5.1: Frequency of bug finding for each case

It is easily observable that the mean of the test group is smaller than that of the control group. However, the question remains on whether the difference is statistically significant such that it can be generalized to the whole population and conclude that the tool adds value for developers. Section 5.2 provides comparison tests between the samples and determines whether they are or not generalizable.

Group		Sample Size	Mean	Standard deviation	Minimum	Q1	Q2	Q3	Maximum
No tool		20	2.05	1.19	1	1	2	3	4
Tool	**Tree visualization**	6	1.67	0.82	1	1	1.5	2	3
	Vertical Partition	10	1.7	1.06	1	1	1	2.5	4
	Sunburst	6	1.83	1.33	1	1	1	2.5	4
	Total	**22**	**1.73**	**1.032**	**1**	**1**	**1**	**2**	**4**

Table 5.2: Descriptive statistics for each of the groups analyzed

5.1 Descriptive statistics

This user study aimed not only to evaluate the performance of the person, but mainly that of the tool. For this purpose, the participants had to rank in a scale from 1 to 5, being 1 the most negative and 5 the most positive, their answers to the following questions:

1. How do you evaluate the user-friendliness of *Crowbar*?
2. Did you use score filtering and, if yes, did you find it useful?
3. Did you find IDE integration important?
4. Did you consider that your programming experience helped you using *Crowbar*?
5. If you did not use the tool, which debugging tools did you use?

For the first five questions, it was evaluated from 1 to 5, being 1 the most negative and 5 the most positive. Descriptive statistics for the first three questions are presented in Table 5.3, while Figures 5.1 and 5.2 are frequency plots for the user-friendliness of *Crowbar* and score filtering, respectively.

Evaluation	Mean	Standard deviation	Minimum	Q1	Q2	Q3	Maximum
Crowbar user-friendliness	4.272	0.528	3	4	4	4.75	5
Usefulness of score-filtering	3.591	1.297	1	3	4	4	5
Importance of IDE integration	4.727	0.456	4	4.25	5	5	5

Table 5.3: Descriptive statistics for questions after user study

To note in Figure 5.1 that only one person (out of 22) gave a score of less than 4 to the user-friendliness of *Crowbar*, contributing to be observed high mean and small standard deviation on the answers to these questions (Table 5.3) and allowing the conclusion that users found the tool user-friendly and intuitive.

Differently, in Figure 5.2, there is more dispersion on the answers, as an observation reflected in the lower mean and higher standard deviation on the second row of Table 5.3. Even though most users evaluated this feature with a 4, the high dispersion in answers may be explained by the fact that this part of the tool is more related to the performance and working method of each user.

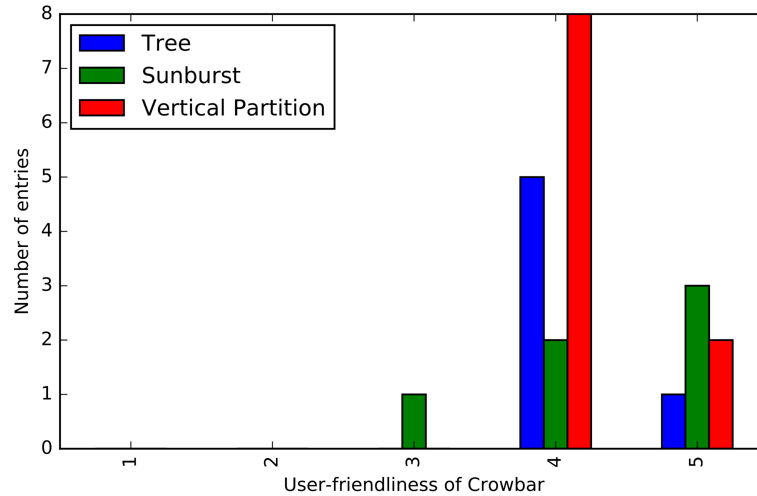

Figure 5.1: Evaluation of user experience by developers that used the tool

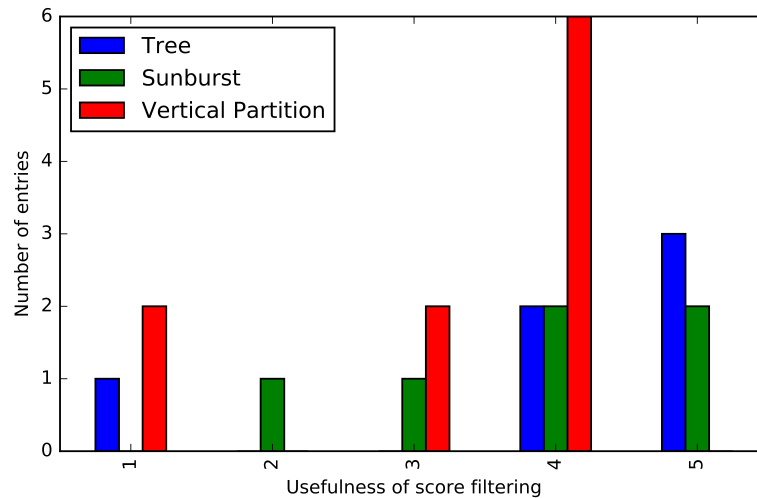

Figure 5.2: Evaluation of usefulness of score-filtering

For example, some users might not need to filter what they see in the visualization to solve the fault (so they did not use it, and gave 1), while others considered it necessary and helpful, giving it a higher value. Given the low mean and the high standard deviation it is not possible to conclude that this new option added value to the users.

Regarding the third question which evaluated the importance of IDE integration, the answers had the highest mean and lowest standard deviation amongst all the questions asked. This fact supports what was stated in Chapter 3 that users tend to use IDEs for debugging and, being this a debugging tool, its integration with the IDE and the possibility to navigate in the code by clicking in the visualization is something that users value when using *Crowbar*.

For those who did not use the tool, they were asked to specify which tools they used to find

the fault. Answers varied among several options. For some of them, it was enough to just read the code as well as the tests to understand the context of the problem and be able to solve the issue. For others, the usage of breakpoints to see each test running step by step as well as monitor the changes in variable values was useful. Some others based their process by adding prints to see the resulting outputs and understand what was the expected return. In this sample, only 4 of them did not find the fault, and from these 4 they used either prints or nothing at all.

5.2 Results comparison

Besides the sample descriptive statistics showed in the previous section, it is also interesting to compare the two groups in other dimensions. As showed before in Table 5.2, the mean time to solve the problem when using the tool is smaller than when no tool is used (1.73 vs 2.05). However, it is important to see whether this samples come from the same population or not such that the conclusions can be generalized. In specific, the two groups should represent different populations such that using the tool leads consistently to lower times than not using the tool. To evaluate this some statistic tests were used, more specifically Mann-Whitney U test[Lun13b] and Kruskal-Wallis H Test[Lun13a].

Mann-Whitney U test is a test used to compare two independent groups with ordinal or continuous dependent variables. For this test to be possible there must be at least 20 elements in each group, which is verified in this case as the group without tool is composed of 20 individuals and the group with tool has 22 participants. For the Kruskal-Wallis H Test, only 5 elements in each group are needed and, therefore, it can be used not only to compare between big groups, but also between the smaller visualization groups created within the "with tool" group which have either 6 or 10 elements each (Table 5.2).

5.2.1 No tool vs Full tool

For the analysis of these two groups, and since there are at least 20 elements in each, Mann-Whitney U test is used.

5.2.1.1 Mann-Whitney U test

For Mann-Whitney U-test the following assumptions need to be satisfied:

- **Dependent variable should be measured at ordinal or continuous level** - Time intervals are mapped from 1 to 4, being 1=0-20 minutes,2=20-40 minutes, 3=40+ minutes and 4= did not find it
- **Independent variables should consist of two or more independent groups** - the groups considered for this test (users who had tool and users who did not have tool) are composed by people who only did one of the experiences on each side, making this assumption also valid.

- **Independence of observations** - those who are in one group cannot be in another one. As stated in the previous assumption, this is also satisfied in this experiment
- **Distributions with the same shape** - as we can see in figure 5.3, the distribution of the values with and without tool have the same shape

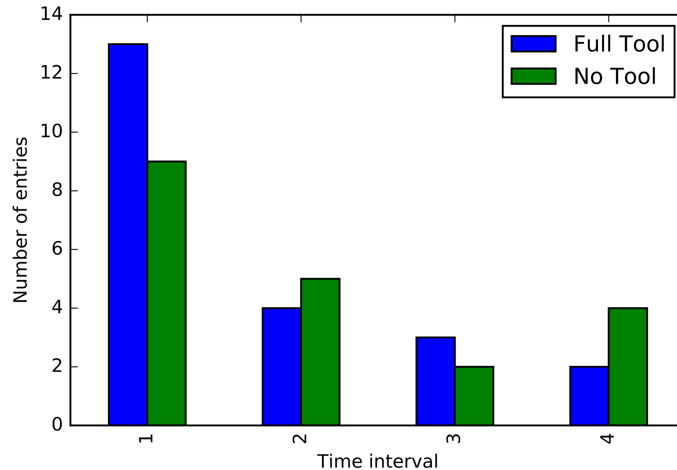

Figure 5.3: Times comparison with and without tool

Having satisfied all the assumptions, it is possible to test the following null hypothesis:

- H_0 - The times' distribution of the two groups are the same
- H_1 - The times' distribution of the two groups are different

To reject H_0 at a 95% confidence level the two-sided p-value the result of p value has to be less than 5%. After doing the calculations, the results are:

Statistic value = 185.5
P-value (both sides) = 0.702

Since the p-value is bigger than 5% it is not possible to reject H_0, and it cannot be said that the times' distributions of having or not the tool are not the same so there is no difference in the times distribution whether they have the tool or not. However, recall that this user study assumes that a big difference would be one of 20 minutes or more. If a smaller granularity was considered for the time samples, with smaller time intervals, it might be the case that the null hypothesis would be rejected. Consider the medians of the two groups reported in column Q2 of Table 5.2. The group with tool has a median of 2 while the other group's median is 1, as the intervals are of 20 minutes this could correspond to a difference of 40 minutes between the two samples, a value much higher than the 1 obtained by using the groups. The problem of having these intervals and not the

exact time duration was identified halfway through the user study, when 20 of the participants had already reported only the time interval and it was not possible to get 20 new participants in order to disregard the first observations.

5.2.2 Comparison between visualizations

Considering the samples for each visualization described in Table 5.2, it is not possible to use Mann-Whitney U test. So, the option goes for Kruskal-Wallis H test which only requires a minimum of 5 elements in each subgroup, a condition that is verified in each visualization. The assumptions for this test are as follows:

1. **Dependent variable at the ordinal or continuous level** - as was already explained in the previous test and as the variable is the same, this assumption is satisfied

2. **Independent variable should consist of two or more categorical, independent groups** - each group is defined as the visualization most used by the user.Three groups were defined: sunburst, vertical partition and tree

3. **Independence of observations** - for this situation, the groups were divided by the visualization each person used the most and the users could not select more than one option. Therefore, it is not possible to have the same person in two different groups.

4. **Distributions with the same shape** - as depicted in Figure 5.4, all three distributions present the same shape.

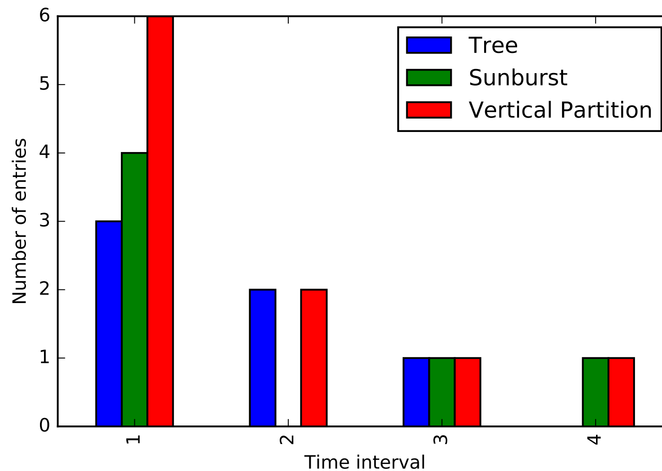

Figure 5.4: Times comparison within visualizations

After observing all these assumptions, define the following hypotheses:

- H_0 - The times' distribution is the same independently of what visualization the participant uses the most

- H_1 - The times' distribution of at least one of the visualizations is different from at least one of the others

Calculating Kruskal-Wallis H test in these groups provided the following results:

$$H = 0.008$$
$$P_{value} = 0.986$$

Again, given the high p-value, it is not possible to reject H_0 as the medians of the three groups are not very different from each other. Independently of the time intervals used that might, once again, bias the results, in fact, the impossibility to reject the null hypothesis as the p-value of almost 99%, this conclusion is also corroborated by the users' comments who mentioned that it is not the use of a specific visualization that makes results better. Therefore, it is safe to conclude that the visualization used does not affect the debugging performance.

5.3 Comparison experience-times

This user study was based in a Java project. Therefore, it is also important to consider if the participants' previous experience with the language affected the amount of time spent to find the issue and correct it. In order to be able to perform Kruskal-Wallis test, only the groups of experience that have 5 or more people are considered. In this case, groups of people with less than one year (2 in total) and more than 4 years (4 in total) are not considered. Table 5.4 shows the distribution of each of the different levels of experience.

Situation	Sample	Mean	Standard Deviation	Minimum	Q1	Q2	Q3	Maximum
1-2 Years	6	1.33	0.816	1	1	1	1	3
2-3 Years	16	1.625	0.957	1	1	1	2	4
3-4 Years	14	2	1.038	1	1	2	2	4

Table 5.4: Frequency of bug finding for each case

5.3.1 Kruskal-Wallis H test

To perform this test, besides having more than 5 elements in each group, it is necessary to satisfy the same assumptions considered in Section 5.2.2, which are also specified below:

1. **Dependent variable at the ordinal or continuous level** - Same variable as the previous tests, so it is satisfied

2. **Independent variable should consist of two or more categorical, independent groups** - Each group is a different level of experience in Java for the user, totalling 3 groups and it is not possible that the same person has two different levels of experience in the language.

3. **Independence of observations** - The same person can only be assigned to the group he/she chose according to his/her experience so, the groups are independent

4. **Distributions with the same shape** - according to Figure 5.5 and to the quartiles in Table 5.4, the distributions do not have the same shape, so this assumption cannot be satisfied

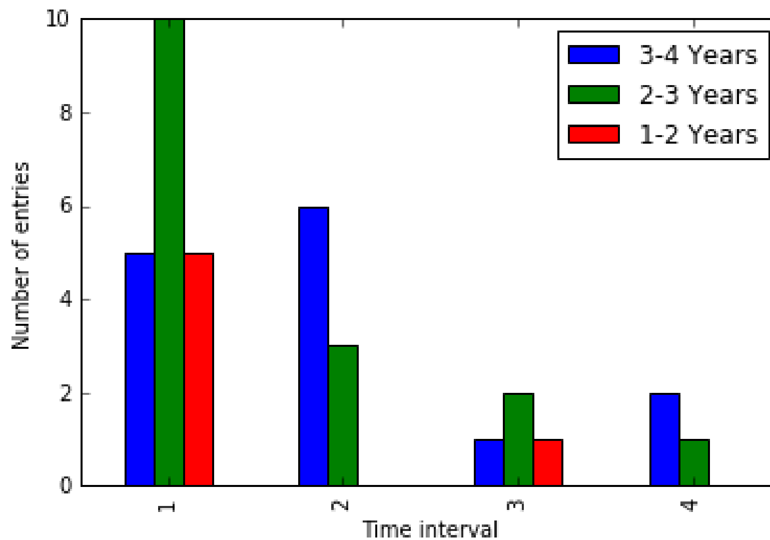

Figure 5.5: Times comparison with different levels of experience in Java

Since only 3 of the 4 assumptions are satisfied, it is not possible to compare the medians of the groups but only their mean ranks. By comparing the mean ranks, it is expected to validate/reject the following null hypothesis:

- H_0 - The times' distribution for all levels of experience is equal

- H_1 - The times' distribution for at least one level of experience is different

It is possible to reject H_0 if p-value is less than 5%. However, by performing the test, the following results were obtained:

$$\text{Statistic} = 3.179$$
$$\text{P value} = 0.204$$

Since p-value is larger than 5%, it is not possible to reject H_0 nor to state that having one more year of experience leads to significantly different results. However, in comparison to the results obtained in Mann-Whitney test for tool versus no tool, the p-value in this case is much smaller than the one just mentioned (0.204 vs 0.702), making it possible to state that the experience that the person has in Java is more important to change the time results than giving them the tool.

5.4 User studies comparison

Before this user study was done, there was already another first study in a previous version of the tool, whose details can be seen in Gouveia's master dissertation [Gou13] as well as in the related article [GCA13]. In both documents, a difference between giving visualizations to users or not when locating faults is documented. However, the conditions of each user study should be considered. While both the user study of this dissertation and Gouveia's conclude that the times' mean and standard deviation of the control group is bigger than those of the test group, the time buffer allowed in each study were different, with Gouveia allowing for only 30 minutes against the 60 minutes from this user study. This implies that people who took more than 30 minutes in this user study would be considered as not finishing in the previous one. By allocating a bigger buffer time, we concluded that more people find the fault without the tool (75% of the control group vs 65% of the previous study). Nevertheless as mentioned before, having or not having the tool is not the only factor affecting performance, since also 9% of the participants with access to the tool did not find it (against 0% in the previous user study). Therefore, it can be concluded that there are other factors affecting how people find faults and understand the code's problems.

5.5 Sum up

This user study aimed to prove whether or not having visualizations and *Crowbar* improved the time spent locating the fault. For this purpose, a code with a fault was sent to the participants, where 22 were asked to solve the problem using the tool (test group) and 20 without the tool (control group). After this, the test group was divided in 3, according to which visualization they used the most. This division was then used to check if one visualization had more impact/benefit for the users than the others.

As for the first test with bigger groups (test group vs control group) Mann-Whitney U test was used since it was not possible to prove that both distributions were normal, but it was visible that they had the same shape. The test was done on whether the samples had or not the same median. By calculating Mann-Whitney U test on both samples, the resulting p-value was higher than 5%, meaning that the null hypothesis could not be rejected and, consequently, it could not be concluded that the medians were not equal.

After, the same type of test was performed for each of the visualizations. In here, it was expected that the medians of the groups would be equal, such that the times would not differ much from one visualization to another. As the number of elements in each group was smaller than 20, the Kruskal-Wallis H test was used. The hypothesis to be tested was whether the medians of the three groups were equal against the alternative that if at least one of them was different. The Kruskal-Wallis H test resulted in a p-value of more than 5%, not allowing for the rejection of null hypothesis, meaning that it is not possible to say that the medians are not equal.

In this study, it was not only considered whether the user had or not access to the tool, but also his/her experience in Java. To test whether experience level affected the performance results,

Kruskal-Wallis tests were ran between each group of experience in Java. While it was not possible to reject the null hypothesis again, the obtained p-value was significantly smaller than the one obtained both in Mann-Whitney for tool vs no tool and in Kruskal-Wallis between each visualization, leading to the reasoning that experience may have a larger impact on the time spent debugging than access to a tool or the type of visualization used.

All these results were then compared with the previous user study done and, even though the granularity of times considered here were bigger than what would be ideal, this study still shows that fault localization does not depend only on the tools given to developers or the experience they have, but also in some other factors that may be evaluated in future work.

Chapter 6

Conclusions and Future Work

6.1 Contributions

This dissertation was composed of three parts: two development parts that took place at the same time and a user study performed later. The first two parts were dedicated to the development in the tool, by investigating and implementing new visualizations and extending the current implementation of the visualizations in order to make it possible to add new ones. In specific, this dissertation contributed by adding to *Crowbar*, the tool in focus in this dissertation, Reingold-Tilford tree visualization equipping it with all the functions already present in the other visualizations of the tool. In parallel with this, the possibility of implementing an IntelliJ IDEA plugin for *Crowbar* was also studied. However, regarding this second topic, the final contribution consisted on describing how it could be done in the future and what kind of tools/libraries could be use to do it.

In the third part, the user study, some conclusions were achieved. The first is that, access to the tool adds value for those who use it. Unfortunately, this conclusion could not be confirmed empirically, namely by testing whether access to it translates into better debugging performance, due to the time intervals chosen. Nevertheless, it is safe to provide a second conclusion that the type of visualization used when debugging does not affect substantially the performance of developers and that the benefits of allowing for score filtering in the visualization depend on the working method of each person. For example, if a developer is very familiar with debugging tools already provided by the IDE (such as breakpoints and step-by-step execution) they will likely only use the basic functions of *Crowbar* to help them navigate through the code not taking advantage of the new features available.

6.2 Answer to research questions

6.2.1 Does having a visual debugging tool for a user result in better performance locating faults?

Even taking into consideration the assumption of big time intervals defined before, it is still possible to observe an increase in the mean time spent debugging a given problem by a person without

access to the tool when comparing to a person with the tool. Therefore, this study points positively for that conclusion, but a new study with detailed data on time spent debugging would help to confirm it.

6.2.2 Does the use of a specific IDE have an impact on their performance?

This was a question in the survey made to people who used the tool. In their opinion, what impacted more their performance was not as much the specific IDE used, but more finding the actual fault and understanding the context of the problem. Again, this supports the argument that the tool(s) used for development is not what affects performance the most.

6.2.3 What kind of tools do people use when locating their faults?

The main factors affecting developers' debugging performance are their knowledge of the code and their experience in the language used in the project. If a person knows well the context of the project he/she is working on, it will be easier to point out what is wrong and what should be changed. Similarly, if he/she has experience in the language of the project, it is easier to try different solutions and see if the bug is solved. Only after these two factors tools reveal to be important. A tool like *Crowbar* can help a small amount of users, provided they know the project, its context and are familiar with the language. If the two first factors are not verified, then the impact of the tool is not as relevant.

6.2.4 What helps people find bugs?

The main factors on finding the bugs in the projects is the knowledge they have in the code and the experience in the language of the same project. If a person knows the context of the project he/she is working on well, then it will be easier to point out what is wrong and what should be changed. Similarly, if he/she has experience in the language of the project, it becomes easier to try different solutions and see if the bug is solved. Only after these two factors tools come into consideration. A tool like *Crowbar* can help a small amount of users, provided they know the project, its context and are familiar with the language. If the two factors do not verify, then the impact of the tool is not as relevant.

6.3 Future work

From this dissertation two main points can be taken for the future:

1. Implement a plugin for IntelliJ IDEA based on Java browser rendering libraries
2. Perform a user study with better granularity in the results to confirm/refuse the conclusions of the user study of this dissertation

References

[AHLW95] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. Fault localization using execution slices and dataflow tests. *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*, pages 143–151, 1995.

[AZV07] Rui Abreu, Peter Zoeteweij, and Arjan J C Van Gemund. On the accuracy of spectrum-based fault localization. Technical report, Delft University of Technology, 2007.

[AZvG07] Rui Abreu, Peter Zoeteweij, and Arjan J C van Gemund. Spectrum-based Multiple Fault Localization. Technical report, Delft University of Technology, Delft, 2007.

[Cle02] Cleanscape. xSlice: A Tool for Program Debugging. Available in `http://www.utdallas.edu/$\sim$ewong/SE6367/01-Project/xsuds-user-manual-html/xslice.html`, last accessed in June 19th 2016, 2002.

[Cro16] Crowbar. Crowbar | For those who hate bugs. Available in `https://www.crowbar.io`, last accessed in February 11th 2016, 2016.

[CRPA12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: an eclipse plug-in for testing and debugging. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012*, page 378, 2012.

[GCA13] Carlos Gouveia, José Campos, and Rui Abreu. Using HTML5 visualizations in software fault localization. *2013 1st IEEE Working Conference on Software Visualization - Proceedings of VISSOFT 2013*, 2013.

[Gee05] David Geer. Eclipse Becomes the Dominant Java IDE. *Computer*, 38(7):16–18, 2005.

[Got05] Greg Goth. Beware the march of this IDE: Eclipse is overshadowing other tool technologies. *IEEE Software*, 22(4):108–111, 2005.

[Gou13] Carlos Gouveia. *HTML5-based Visualizations to Support Software Fault Isolation*. Master thesis, Faculdade de Engenharia da Universidade do Porto, 2013.

[JH05] James A Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. Technical report, College of Computing, Georgia Institute of Technology, 2005.

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. *Issta*, pages 437–440, 2014.

[Lun13a] Lund Research Ltd. Kruskal-Wallis H Test in SPSS Statistics. Available in `https://statistics.laerd.com/spss-tutorials/`

kruskal-wallis-h-test-using-spss-statistics.php, last access in June 8th 2016, 2013.

[Lun13b] Lund Research Ltd. Mann-Whitney U Test in SPSS Statistics. Available in https://statistics.laerd.com/spss-tutorials/mann-whitney-u-test-using-spss-statistics.php, last accessed in June 21st 2016, 2013.

[MKF06] G.C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[NDe16] NDepend. Improve your .NET quality with NDepend. Available in http://www.ndepend.com/screenshots, last access in June 21st 2016, 2016.

[Phi07] Friedrich Steimann Philipp Bouillon, Jens Krinke, Nils Meyer. EzUnit: A Framework for Associating Failed Unit Tests with Potential Programming Errors. *Agile Processes in Software Engineering and Extreme Programming*, 4536:101–104, 2007.

[RT81] E.M. Reingold and J.S. Tilford. Tidier Drawings of Trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, 1981.

[SR83] Kenneth J. Supowit and Edward M. Reingold. The complexity of drawing trees nicely. *Acta Informatica*, 18(4):377–392, 1983.

[s.r16] Jetbrains s.r.o. IntelliJ IDEA the Java IDE. Available in http://jetbrains.com/idea/, last time accessed in June 21st, 2016, 2016.

[Tea16] TeamDev. JxBrowser - a Chromium-based Swing/JavaFX component. Available in http://www.fernuni-hagen.de/ps/prjs/EzUnit4/, last accessed in June 21st 2016, 2016.

[The16] The Eclipse Foundation. SWT: The Standard Widget Toolkit. Available in https://www.eclipse.org/swt/, last accessed in June 21st 2016, 2016.

[WD09] We Wong and Vidroha Debroy. A survey of software fault localization. *Department of Computer Science, University of ...*, (November):1–19, 2009.

[WL08] Richard Wettel and Michele Lanza. CodeCity: 3D visualization of large-scale software. *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 921–922, 2008.

[ZL96] Andreas Zeller and Dorothea Lütkehaus. DDD—a free graphical front-end for UNIX debuggers. *ACM SIGPLAN Notices*, 31(1):22–27, 1996.