FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

GZoltar: A Graphical Debugger Interface

André Riboira

Master in Informatics and Computing Engineering

Supervisor: Rui Maranhão (PhD)

Co-Supervisor: Rui Rodrigues (PhD)

17^{th} January, 2011

GZoltar: A Graphical Debugger Interface

André Riboira

Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: João Carlos Pascoal de Faria (PhD)
External Examiner: António Nestor Ribeiro (PhD)
Supervisor: Rui Filipe Lima Maranhão de Abreu (PhD)

11^{th} February, 2011

Abstract

Locating components which are responsible for observed failures is the most expensive, error-prone phase in the software development life cycle [HS02]. Automated diagnosis of software faults (aka bugs) can improve the efficiency of the debugging process, and is therefore an important process for the development of dependable software.

In the past, we have presented a toolset for automatic fault localization, dubbed Zoltar, which adopts a fault localization technique based on abstractions of program traces [Abr09, JHS02a]. The toolset [JAG09] provides the infrastructure to automatically instrument the source code to produce runtime data, which is subsequently analyzed to return a ranked list of potential faulty locations. Using a thread-based example program as well as a large, realistic program, we show the applicability of the proposed toolset in [JAG09].

Although its output is deemed useful [Abr09, JHS02a], Zoltar's debugging potential has been limited by the lack of a visualization tool that provides intutitive feedback about the defect distribution over the code base, and easy access to the faulty locations. To help unleash that potential, we propose exploring two visualization techniques - treemap and sunburst - aimed at aiding the developer to acquire a broad sense of the error distribution, and find faults quickly.

The visualizations are implemented as an Eclipse [Gee05] plugin, dubbed GZoltar, allowing direct access from the visualization tool to the faulty locations. Eclipse has been chosen because it is a popular integrated development environment. Compared to other visualization techniques, such as [JAG09, ZL96, JHS02a], our tool shows the relevant information for the debugging process in a more intuitive way. Experiments with human developers show that the technique has indeed the potential to aid in locating software faults.

Acknowledgments

This project could not have been accomplished without the help of several people. I certainly will forget to mention many people who helped me in some way, so I want to start by doing a generic acknowledgment to all those who helped me during these last days.

I can not forget to make a special acknowledgment to Prof. Dr. Rui Maranhão and to Prof. Dr. Rui Rodrigues. Since the beginning that they are always extremely helpful, and motivated me not only to believe in my work and ideas, but also helped me on all those "gray days", by lighting up my mind and showing the right direction to overcome all the barriers. They are much more than mere mentors (and no, they are not "tormentors"!), they are now two friends. All this support led me to not even consider to leave my academic life, and encouraged me to proceed to the next level (PhD). I would like to mention my gratitude also to Prof. Dr. Francisco Restivo, for its valuable help during my dissertation preparation, to Prof. Dr. Augusto Sousa, Prof. Dr. Raul Vidal, Prof. Dr. Eugénio Oliveira, Hugo Ferreira, and so many other teachers and researchers of the Faculty of Engineering of the University of Porto, and to Prof. Dr. João Saraiva and many other teachers and researchers of University of Minho. I would also like to mention my gratitude for my laboratory colleges, Prof. Dr. João Pascoal Faria, Prof. Dr. Ana Paiva, Francisco Andrade, and all the others, for every little thing that they did to help me during this journey. Because this MSc would not be possible without my academic and professional background, I cannot forget to mention also my gratitude to Prof. Dr. António Costa, Prof. Dr. Constantino Martins, and all my former teachers in the School of Engineering of Polytechnic of Porto and to Prof. Dr. Altamiro Pereira, Prof. Dr. Alberto Freitas, Eng. Jorge Gomes and Tiago Costa from Faculty of Medicine of University of Porto.

I believe that life is a system that should be well balanced to work properly. I have no doubt that without the help of all my family and friends, I could not accomplish this project. Therefore, I want do make a special acknowledgment to my parents, Jerónimo and Natália, that supported me since day 1 (well, in fact they did supported me since month -9), and to my super-brothers Nuno and Hugo, and their girlfriends Patricia and Gisela. I also want to thank to the rest of my family, specially to my super-cousin Cláudia and her boyfriend Helder, that are like brothers to me (and yes, they are all "super"). A want to thank also to my future parents-in-law and to my future brother-in-law, for all the help. Now just a little sentence in Portuguese: "Muito obrigado a todos, vocês são os maiores!". My friends also supported me in every moments, the good and the bad ones. I am very grateful to all of them. I will not enumerate them because they are so many and I could forget to mention some of them. Again, a Portuguese moment: "Muito obrigado pessoal! Vocês foram todos impecáveis!". Last but obviously not least, I would like to thank my *fiancée* Raquel, for being the most wonderful person of my life. Without her support and comprehension I could never finish this journey. "Obrigado por tudo amor!"

Porto, January 16, 2011

André Riboira

"To accomplish great things we must first dream,
then visualize, then plan... believe... act!"

Alfred A. Montapert

Contents

CONTENTS

List of Figures

List of Tables

List of Acronyms

ACM Association for Computing Machinery

API Application Programming Interface

AWT Abstract Window Toolkit

CMMI Capability Maturity Model Integration

CPU Central Processing Unit

DDD Data Display Debugger

ESA European Space Agency

ESI European Software Institute

FEUP Faculty of Engineering - University of Porto

GPGPU General-Purpose computation on Graphics Processing Units

IEEE Institute of Electrical and Electronics Engineers

IDE Integrated Development Environment

ISTQB International Software Testing Qualifications Board

ITIL Information Technology Infrastructure Library

JOGL Java OpenGL

MAPi MAP Doctoral Program in Computer Science

MSc Master of Science

OpenGL Open Graphics Library

OSGi Open Services Gateway initiative

PDE Plug-in Development Environment

PhD Doctor of Philosophy

PMBOK Project Management Book of Knowledge

RUP Rational Unified Process

SEI Software Engineering Institute

SFL Spectrum-based Fault Localization

SIL Safety Integrity Level

SUT System Under Test

SWT Standard Widget Toolkit

TAIC-PART International Conference on Testing: Academic & Industrial Conference Practice and Research Techniques

TSP/PSP Team Software Process / Personal Software Process

TUDelft Delft University of Technology

UI User Interface

VM Virtual Machine

XP Extreme Programming

Chapter 1

Introduction

Visualize (Pronunciation: /'vɪʒjʊəlʌɪz, -zj-/) *verb*

1. form a mental image of; imagine

2. make (something) visible to the eye [Pre11]

GZoltar is an Eclipse-based plug-in that aims at helping the software debugging task, which is one of the most expensive tasks of the software development life-cycle [HS02]). It aims at allowing its user (typically a software developer) to *form a mental image of* the faulty System Under Test (SUT), to *imagine* it, to make a conceptual representation of that system *visible to the eye*. Shortly, GZoltar wants to offer a powerful visualization of the SUT, this way aiding developers to find the root-cause of software faults quickly. For debugging tasks, that visualization should not only present the SUT's hierarchical structure (the way code is organized, from a grain-size to coarse-size level of abstraction), but also display helpful information about results of automatic debugging processes. If a user has a powerful visualization of the SUT hierarchical structure, and can also quickly see which system components have a bigger failure probability, he will certainly localize faults faster. If added to this he has the ability to see how components correlate with each other, he has certainly an helpful tool to reduce debugging task time, and therefore time-to-market.

1.1 About Debugging

Bugs exist since the beginning of computer science. There has been much speculation concerning the early use of the word "bug" in the domain of computer science. Usually it is referred as the first bug in computer science history the one found on the Harvard University Mark II Aiken Relay Calculator, in 1947 (see Fig. 1.1). That machine was experiencing problems and operators found a moth trapped on the system while investigating the source of the system's malfunction. The bug was affixed to the log, and the term "debugging" was born (referring to fix faults in computer systems) [Coh94]. More important than the first bug, are the bugs that still exist on current software applications.

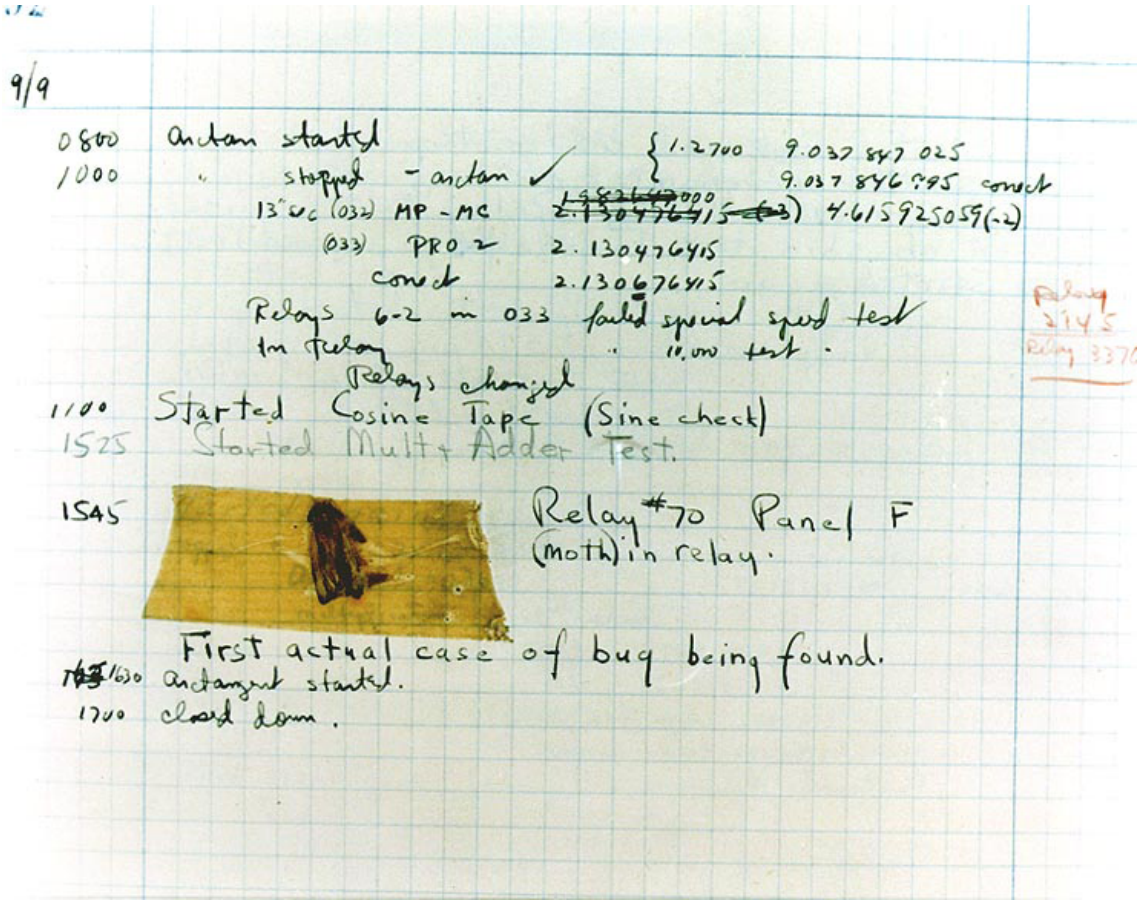

Figure 1.1: **Possibly the first bug on a computer system.** Operators log, with the possible first bug in a computer system affixed on it.

A software bug is nothing more than a fault [1], that when executed will give a result different than the one it was supposed to give, i.e. output does not follow the specification [Avi71]. Every large software project has bugs, and many are difficult to find, because there are faults that are only exercised on odd combinations of factors. Sometimes, a developer also thinks that an error is caused by a given fault, and in fact it is caused by another one, in a completely different location. Moreover, in large projects, software fault localization is even more difficult. Sometimes the person who is looking for the fault is not even the software developer of that given module, and a "step-by-step" analysis of the system behavior turns into a very time-consuming, cumbersome task, because he has to understand the software implementation's details. In addition, developer also can try to make recursive isolations to find the location of the bug. First it is necessary to find out what module is causing the error, and then, its faulty submodule, and so on. With completely independent modules this can be done fairly well, but it is particularly hard if the modules have several dependencies of other modules. In that case, developer reaches to a situation almost identical to a "step-by-step" analysis, because he cannot analyze the modules like independent entities, he has to analyze the system as a unique entity. All these steps can take too long, and can be a

[1] Faults/defects/bugs are used interchangeably in this thesis

very resource-consuming task. This is one of the reasons why the fault localization process is so expensive. There are some safety critical systems where this is a very big issue. Although there are already powerful debugging tools, in general they are not widely used. The main reason is probably because they are not very easy to use, and their results are not easy to understand. At this moment, Zoltar [JAG09] is the software fault localization tool that delivers the most efficient diagnostic results [AZGvG09], but like most of the debugging tools, it lacks an intuitive interface.

1.2 Motivation

This section has the project motivation. It has a brief description about software faults, safety critical systems and the relation between faults and accidents. It also has the main problem of debugging tools: the lack of a powerful graphical debugger. There are already helpful tools and concepts, but no powerful graphical debugger tool, and that is in fact the main motivation to create GZoltar Project, precisely to fill that gap.

1.2.1 Safety Critical Systems

There are many software-dependent systems where a fault can cost hundreds of lives or an environmental disaster. A system failure can have a very distinct impact, depending on the nature of the system itself [DA09]. If considered a personal computer system crash, the damages are minimal when compared for instance with an aircraft system crash. While in a personal computer, a system crash may represent some loss, due to the temporary system unavailability, or some data loss, on an aircraft a system crash may represent a crash of the whole airplane (see Fig. 1.2 for more examples of Safety Critical Systems). It is obvious that safety critical systems must have mechanisms of tolerance to errors, but the efficiency of those mechanisms is directly related to the quality of the software that is implemented [LBK90]. Some faults remain unnoticed in a production system for a long time, until they are finally activated, causing an error. If the system was not designed to tolerate that error, it will cause a system failure, and this may cause major accidents.

1.2.2 From a Fault to an Accident

When defects are activated, they cause system errors. Those errors can lead to some serious system malfunctions (system failure), which may unfortunately lead to accidents [ALRL04]. A fault in software, for itself, does not represent an hazard. Obviously, if that code is never executed, the fault does not cause any problem to the system. However, if all the conditions are met, the fault can be executed, causing a system error. An error is a different result from the one the system was supposed to give. Every time a faulty code is run, an error is produced [Avi71]. The system can have resources to deal with some errors, in order to tolerate them, but if it cannot do it (because it was not considered or there were too many errors that hinder the recovery), the system will fail. This can lead to an accident if there where loss of lives or an environmental or economical disaster,

Figure 1.2: **Some examples of contexts where Safety Critical Systems are involved.** Those systems must not only have a minimum amount of faults, but also be tolerant to some possible error, to avoid a system failure.

or to an incident, if there is no loss, but could hypothetically has in a similar situation (see Fig. 1.3 for a diagram explaining the sequence from a fault to an accident).

Although safety critical systems are designed to tolerate errors, they depend considerably on the software quality, not only of the modules themselves but also of the error recovery logic implementations. Software Safety Integrity Level (SIL) are also obtained considering the probability of failure of a given system. That probability will vary depending on the number of faults present in that system [BH08].

It is possible to enhance system safety by correcting its faults and develop it in an error tolerant approach. This project is about the first and most difficult step related to fault correction: finding its localization. To correct a fault it is needed to know its exact localization. There are some tools to help us in that task, but there is a lack of integration and user interaction which can lead not only to greater difficulty in their adoption, but also to a greater difficulty in its use and in obtaining and understanding its results [SMBM06].

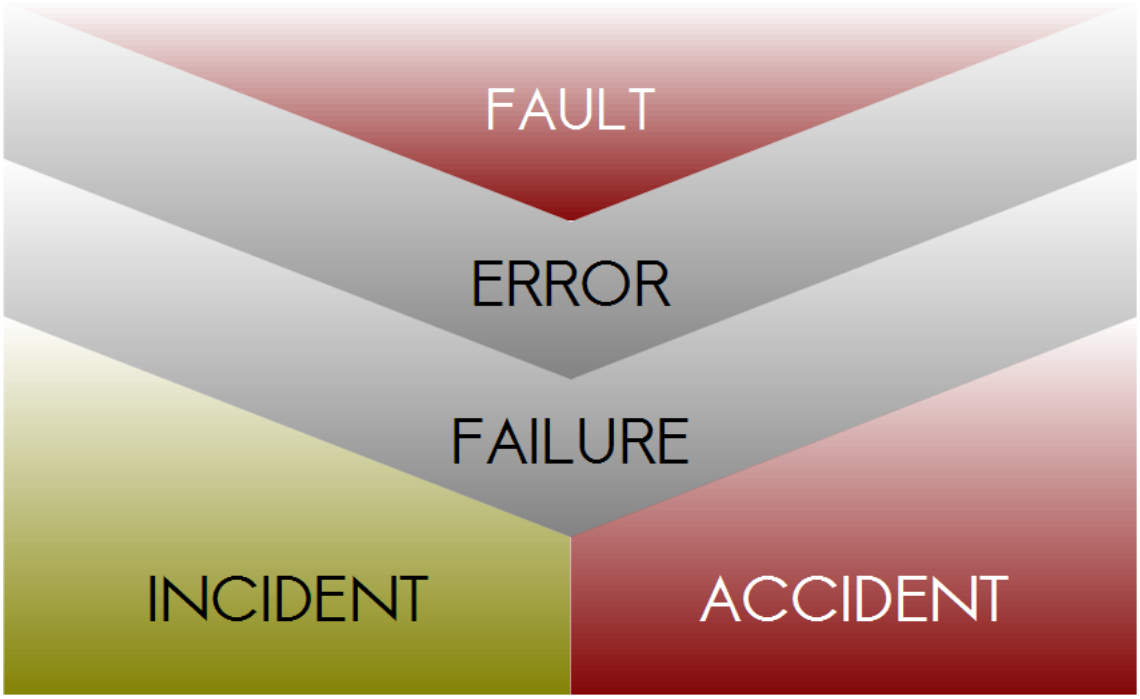

Figure 1.3: **From a Fault to an Accident.** System generally has faults, and when they are activated, an error is produced. If the system does not tolerate that error, it will fail, what can lead to an accident when on Safety Critical Systems (or an incident if there were no loss).

1.2.3 Visual Debugging Tools

Currently available graphical debugging tools do not provide an integrated environment that allow developer to localize and correct software faults at the same place. This forces the programmer to switch constantly between applications, what will certainly lead to a loss of productivity. There are tools that are integrated in an Integrated Development Environment (IDE) but does not offer a powerful visualization. External tools are quite powerful, but it lacks integration with other tools and mainly with an IDE. There is a clear opportunity to create a new tool, witch fills the current gap of graphical debugging tools, and that is the main motivation of this project. The idea of creating an useful and innovative tool is very attractive.

1.3 Main Goals

This project aims to develop an advanced user interface for fault localization, with a powerful project view, that uses Zoltar and other useful debugging tools as input. To enhance user comfort, it was be developed as an Eclipse plug-in because it is one of the most popular IDE nowadays [Gee05]. The first idea was to create GZoltar as a simple viewer to display Zoltar's output. While this functionality was needed, certainly it is possible to go further and create something more powerful, that enriches the way the programmer sees his project, and all of this integrated in his current work tools.

The main goal of this project is to **create an application that can provide a quick view of a project structure, the relations of its lines of code and the probability of each component to fail**.

There is an hierarchical localization view, where a component is connected to its child-components (sub-components) or its parent ones. It is possible to have a clear perception about the level of that component (from a wider to a narrower view). There are many ways to represent this kind of information, and another good one to represent it is a dynamical view, where the viewer can "zoom-in" to inside that node, where he will find it's sub-components. It is also possible to "zoom-out" to have a progressive "big picture" of the project (see Fig. 1.4 for a graphical representation of this concept). The graphic representation should be able to provide the maximum amount of information to the viewer, and he should be able to navigate easily through it.

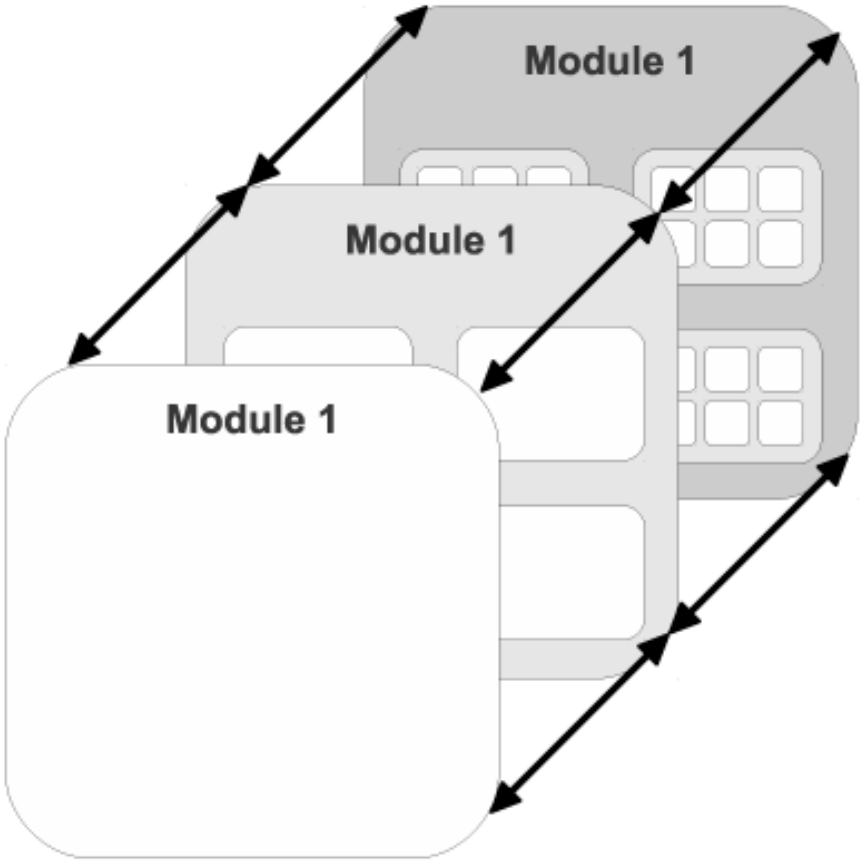

Figure 1.4: **GZoltar's Hierarchical View.** One of GZoltar's goal is to provide an hierarchical view of the SUT. It should allow the possibility to navigate between different hierarchical levels, like a conceptual zoom, from a brief view of the system to a more detailed one.

1.4 Document Structure

In this section will be presented this document structure. Apart from Introduction, this report has four more chapters.

Chapter 2 contains the state of the art in this project field.

Chapter 3 describes in detail the architecture of GZoltar Project.

Chapter 4 details case studies made to evaluate some dynamic visualizations processed by GZoltar Project.

Chapter 5 has the conclusions about this project, and also some ideas for future work.

Chapter 2

State of the Art

Debugging can be split into two main categories:

- **manual debugging tools**, that are based on a step-by-step execution of a single test case to try to localize the system fault;
- **statistical debugging tools**, that are based on historical information about multiple test executions (and obviously on the test execution result).

Within these two categories, there are:

- the ones that **integrates into an IDE**;
- the **standalone tools**.

Each tool has its advantages/disadvantages. This chapter presents the state of the art of graphical debugger interfaces.

2.1 Fault Localization Techniques

As said before, fault localization is an existing task since the beginning of computing and many tools have been created to assist in this process. There are traditional debugging techniques, where a software developer tries to localize the fault by executing each test at a time, and analyzing the system behavior during those tests, and there are statistical debugging techniques, that are based on an history of test results, and where is applied some algorithm to calculate the failure probability of each software component.

2.1.1 Traditional Debugging

Manual Debugging is a process to localize software faults by analyzing an execution of that system step-by-step. Only one execution at a time is considered. Usually when a software tester knows some test cases that he knows that produce system errors, he execute them step-by-step, analyzing

the system, to try to isolate the faulty component, and correct its fault. This process is usually better for small system parts, and when a test case can be easily reproducible.

Although there are already tools that provide automatic fault localization, they are not widely adopted. Software developers tend to use manual debugging tools to find their software faults [JAG09]. The most probable reason for that may be the fact that automatic debugging tools are generally not easy to use, and their output is not easy to understand and analyze. The lack of integration between all available tools and IDE's may be also another strong reason for their lack of adoption by software developers [SMBM06].

2.1.2 Statistical Debugging Tools

There are some applications that uses statistical techniques to calculate software components failure probability, based on different approaches. One of the most effective statistical technique is Spectrum-based Fault Localization (SFL), that uses historical execution data of a given system, having also the result of each execution (if it was successful or not). In SFL, an input matrix is built, having the information of what components were used at each test execution, and that test result, as an error vector (see Fig. 2.1). This matrix is the input of all SFL algorithms. Each one of the main matrix column represents a SUT component, and the column of the last one represents the error vector, with the information if the execution has passed or not. Each line of these matrices represents a execution of a test case. This is the only information a SFL algorithm has as input, to calculate the failure probability of each component (matrix column). Having this input data, is then executed one of the many SFL algorithms to calculate the failure probability of each component (each column of the input matrix).

$$
\begin{array}{l}
\\
N \text{ spectra}
\end{array}
\begin{array}{c}
\overbrace{\phantom{a_{11}\ a_{12}\ \cdots\ a_{1M}}}^{M \text{ components}} \\
\begin{bmatrix}
a_{11} & a_{12} & \cdots & a_{1M} \\
a_{21} & a_{22} & \cdots & a_{2M} \\
\vdots & \vdots & \ddots & \vdots \\
a_{N1} & a_{N2} & \cdots & a_{NM}
\end{bmatrix}
\end{array}
\quad
\begin{array}{c}
\text{error} \\
\text{detection} \\
\begin{bmatrix}
e_1 \\ e_2 \\ \vdots \\ e_N
\end{bmatrix}
\end{array}
$$

Figure 2.1: **SFL input matrix.** N means test executions, M means SUT components, a means code coverage and e means test execution result.

2.2 IDE Integrated Debugging

A software developer tends to use tools that are more comfortable to him. Usually, software is developed in some IDE, which provides a lot of useful tools that helps him during software development. Those tools can give not only useful functionalities about code editing, like line numbers and syntax highlight, but also about project organization, code completion, integrated help and the ability to make breakpoints and see the system state at a given stage. It is also possible to do a step-by-step execution to analyze system behavior at each line. Unlike IDE's,

where many tools interact together in the same environment where the programmer edits his own source code, automatic software debugging tools tend to be standalone solutions.

2.2.1 Breakpoints

IDE's offer only basic debugging capabilities. It is possible to make some breakpoints to allow a step-by-step execution, while analyzing the system state (variable and memory position values, for instance). This is a good technique to debug a system where is known exactly how to reproduce the system error. For general debug of large projects, this is not the best way to debug, because it is a very time-consuming task. For large projects one should consider the use of automated debugging techniques, but unfortunately they are not commonly integrated into any popular IDE.

2.2.2 Conditional Breakpoints

A Conditional Breakpoint is similar to traditional breakpoints, but they are only activated on certain user-defined circumstances. This allows the user to create filters to accelerate debugging process.

2.2.3 Assertions

Assertions are programming techniques where developer create assertions during code [Ros95]. When developer expects some value in a certain part of its software, it can "assert" it, so if in any case, during system execution, that assert is not verified, the fault is then immediately detected.

2.2.4 Profiling

Profiling is the ability to obtain the time spent in the execution of each software component, as well as the information about which functions called which other functions. With this information it is possible to know witch software components are slower that expected. It is also possible to know which functions are called more or less often than expected. One of the most well-known profiling tool is GNU's gprof [GKM82]. Eclipse also has a profiling plug-in available: MEProf [HBK04].

2.2.5 Code Coverage

A way to make an analysis of an entire system is to use code-coverage tools [BWK07]. These tools allow us to see which lines were executed during a system test run. With this information, it is possible to see if the program is working as expected, and if not, which lines where involved in the system failure.

Code coverage is already available for some IDE's. For Eclipse, there is EclEmma, a plug-in that makes code-coverage for many executions [Hof11b]). It uses different colors in each code line to show if it was executed or not, but does not calculate its failure probability (see Fig. 2.2). This Eclipse plug-in highlights the lines of code in the code editor with colors that represents if a line was fully executed (green), partially executed (yellow) or not executed (red). It also includes

a view in a form of a tree list, which has a hierarchical structure of the SUT and their coverage. However, this plug-in does not show the failure probability of each component, only if a line of code was covered or not by the test executions. There are also other code coverage tools for Eclipse IDE and other IDE's. Latest versions of Microsoft Visual Studio has code coverage capabilities integrated on its base package [LCBR05].

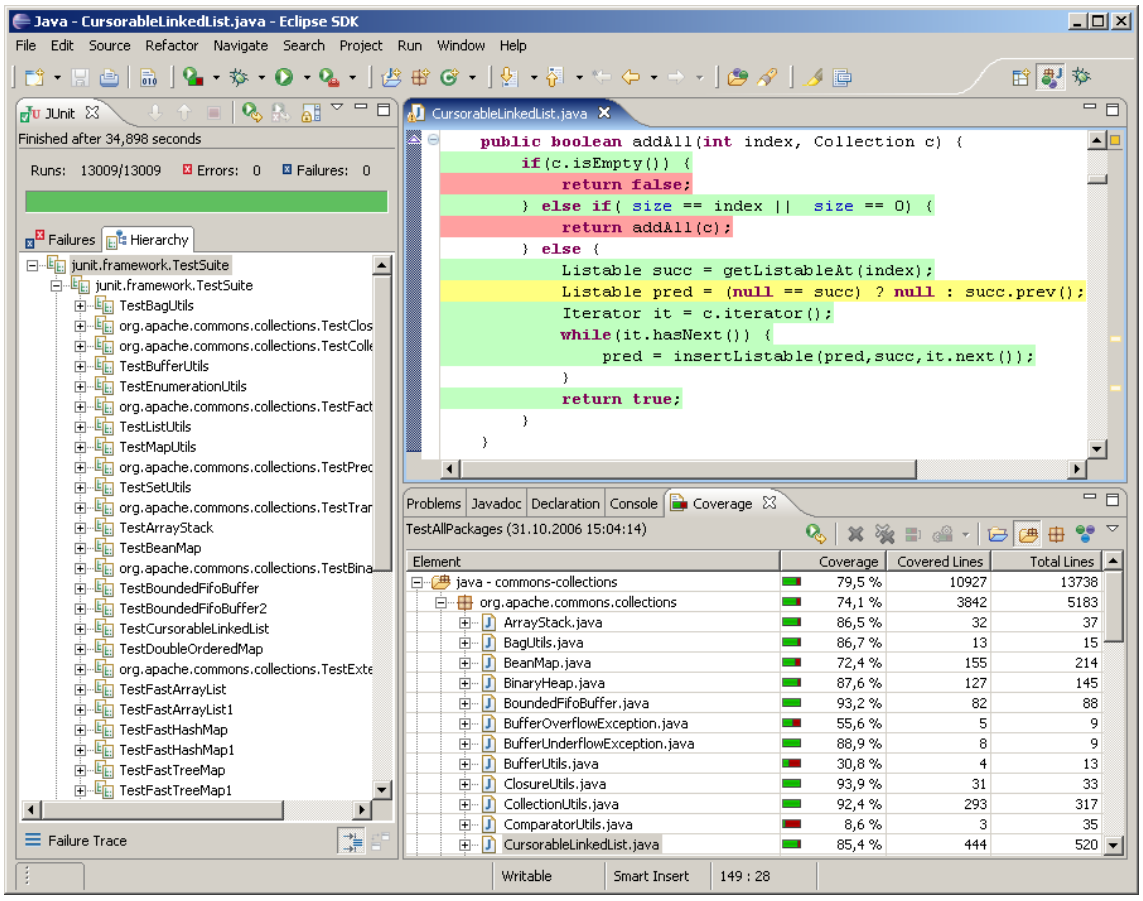

Figure 2.2: **EclEmma Interface.** It offers line highlight reflecting code coverage and a tree list with each SUT component.

2.2.6 Statistical Debugging

There are some tools that integrates into IDE's, that implements statistical debugging. On of these tools is EZUnit. It has a view with a list of code blocks and their failure probability. Each line of that list is highlighted with a color that represents the block failure probability (see Fig. 2.3 for a demonstration of such a view). EZUnit also offers the ability to display a view with a call-graph of a selected test case (see Fig. 2.4 for an example of a call-graph view). This tool integrates also into the code editor, by placing marks on each line, with information about the failure probability of that code block (see Fig. 2.5 for an example of integration between EzUnit and code editor).

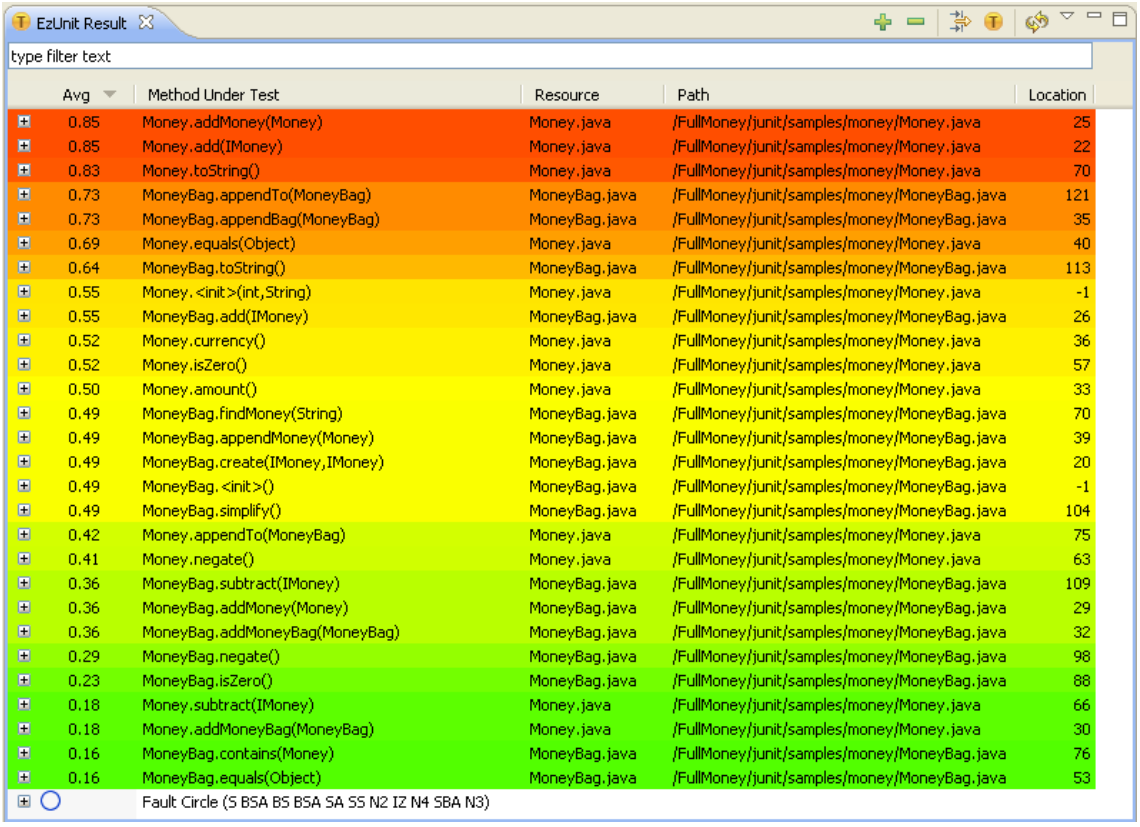

Figure 2.3: **EzUnit view.** This Eclipse plug-in offers a view with a list that has the SUT components highlighted by its failure probability value.

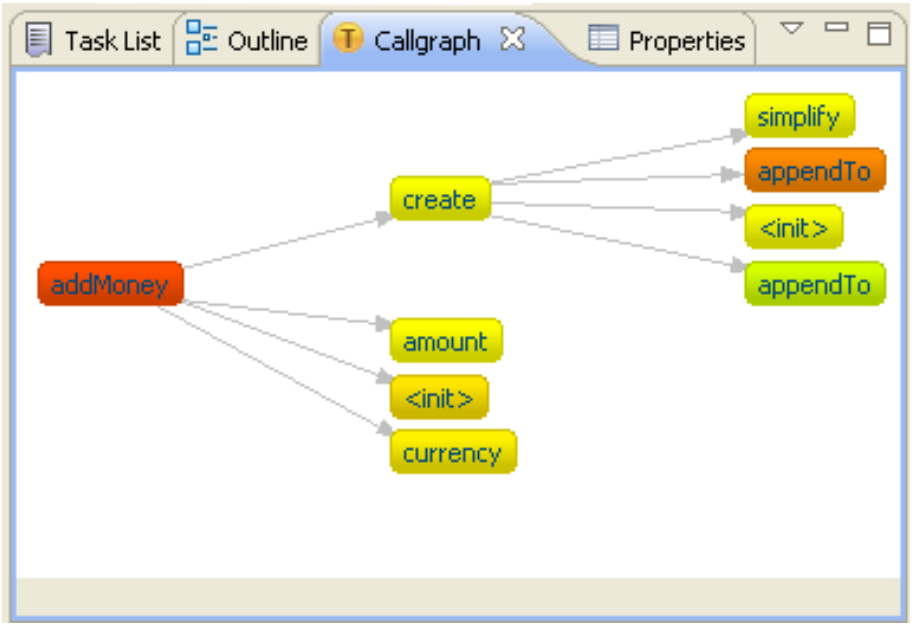

Figure 2.4: **EzUnit Call-graph View.** It is possible to visualize a graph representing the calls between methods of a selected test case.

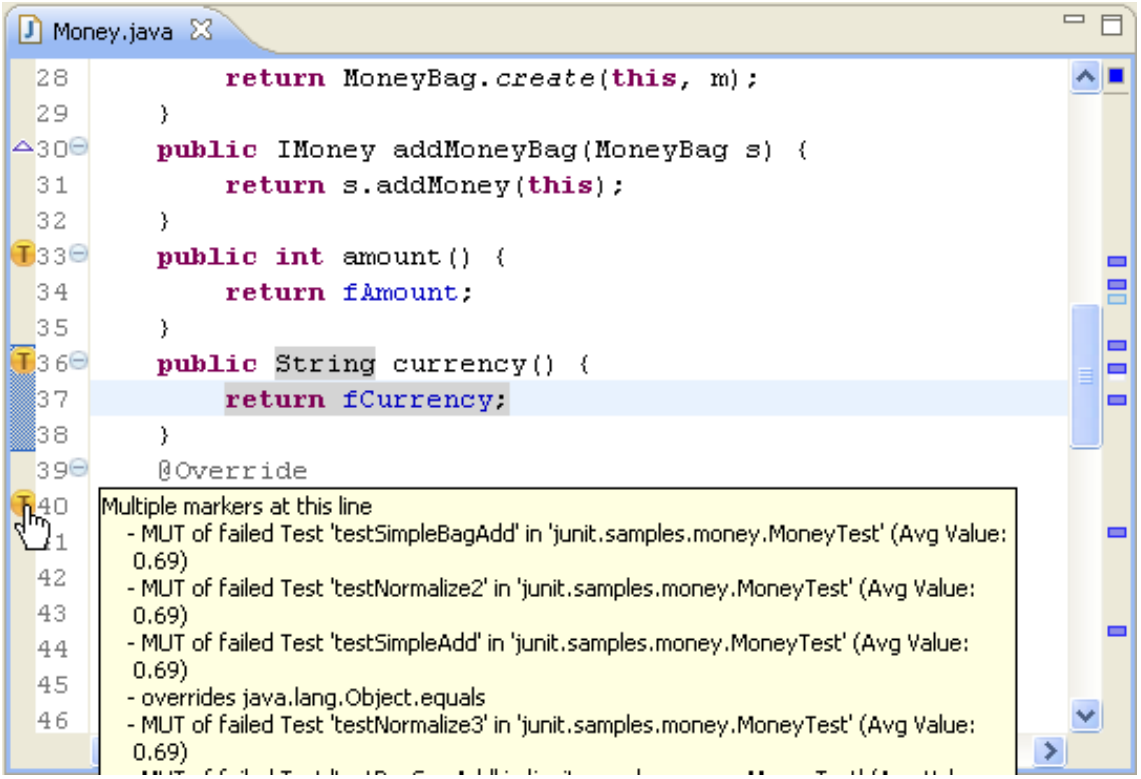

Figure 2.5: **EzUnit Eclipse Integration.** This Eclipse plug-in also integrates with code editor. It offers tooltips with information about that line failure during test executions.

2.3 Standalone Visual Tools

Currently, the most powerful debugging tools are external to any IDE. This, allied to their poor graphical interface, adversely affect their adoption by programmers. However, those tools are very powerful and their use increases considerably the efficiency of fault localization process.

2.3.1 DDD

There are already some tools that provide true graphical debugging capabilities. Data Display Debugger (DDD) is a debug tool that helps you find the software failures through sequential analysis of its implementation. DDD offers an analysis for each software execution, like a traditional approach to the debug process. It allows a step-by-step visual trace of the software execution, not a global analysis of the whole system (see Fig. 2.6 for an example of a DDD visualization) [ZL96].

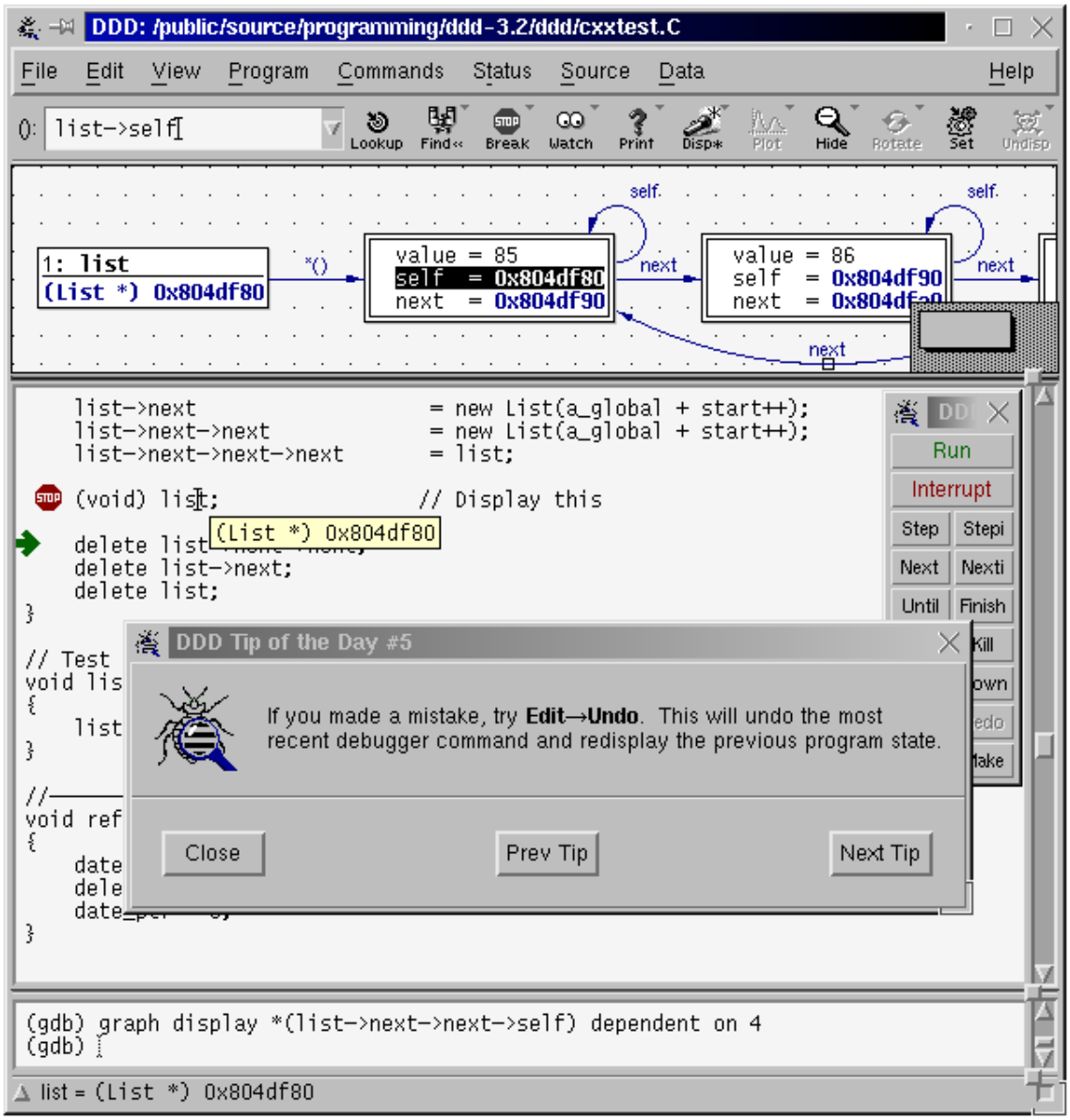

Figure 2.6: **DDD Interface.** At the top the window has a representation of the evolution of system states. It is possible to analyze the system execution visually, in a step-by-step approach.

2.3.2 Tarantula

Tarantula [JHS02b] is the current reference in visual debugging. This tool offers a view where the developer can see all the project code lines at once, highlighted with the fail probability of each line (see Fig. 2.7 for an example of such a visualization). Tarantula is based only on code coverage analysis, and shows a map with all the code lines of the project at once, using different colors in each line to show the probability of each line being faulty. Tarantula makes possible to analyze the whole system at once, without analyzing it step-by-step but with a set of test executions. This is particularly convenient to test large projects.

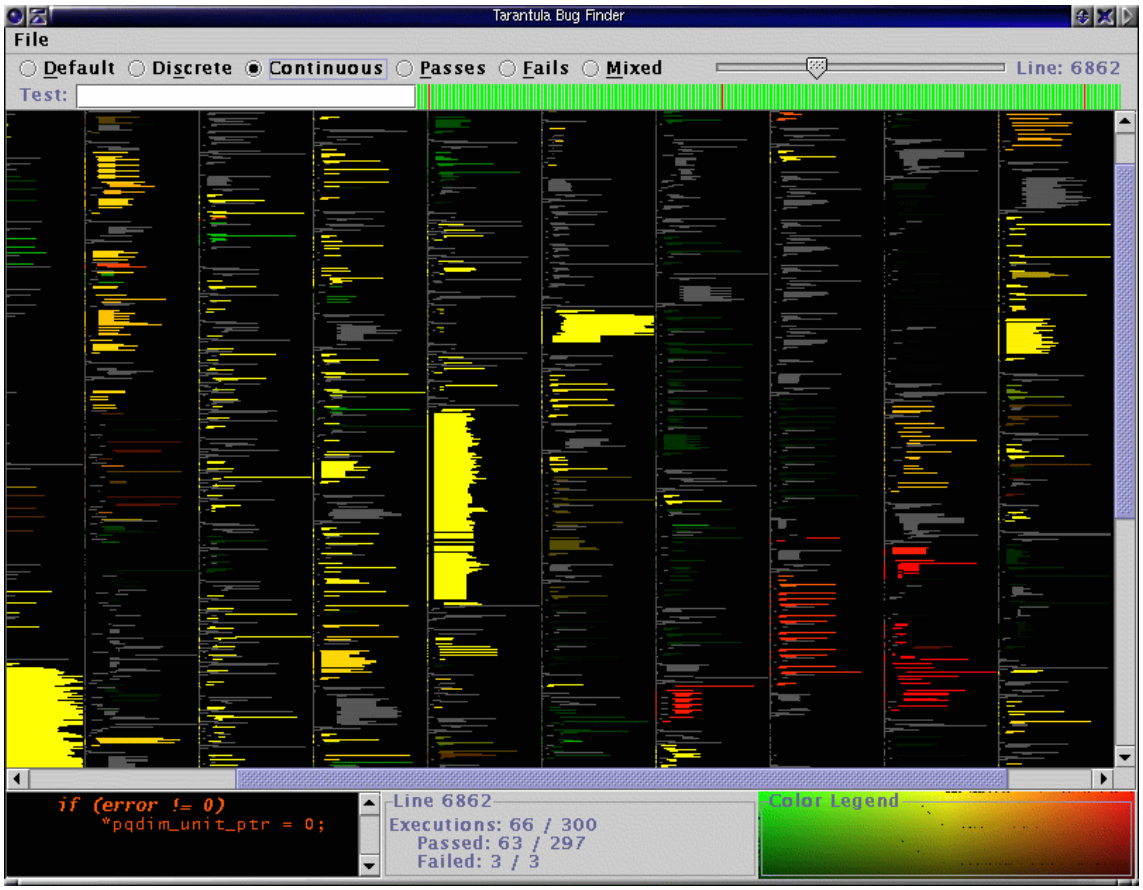

Figure 2.7: **Tarantula Interface.** This visualization is based on lines of code coloring, by their failure probability, varying from red (maximum) and green (minimum failure probability).

2.3.3 Zoltar

Zoltar [JAG09] is a tool that implements SFL algorithms, is written in C/C++, and was created to work with C/C++ projects. Nevertheless, that tool can be ported easily to other platforms, because it has a direct implementation of SFL algorithms, and does not require any special library, platform, or framework to work. It presents superior performance when compared to other similar tools, and it has been proved effective in many fields [AZGvG09].

Zoltar receives a code coverage matrix and an error vector (see Fig. 2.1) as input. It executes its SFL algorithm, and produces a vector with the failure probability of each component. Simplifying:

$$(A, e) \rightarrow Zoltar \rightarrow D$$

Figure 2.8: **Zoltar's Input and Output.** A is a code coverage matrix, e is an error vector and D is a failure probability vector.

2.3.3.1 Performance

Currently, one of the most effective technique to localize software faults is SFL like the one implemented on Zoltar. Zoltar implements Barinel, Tarantula and Ochiai algorithms. In particular,

Barinel and Ochiai algorithms have proved to be the most efficient in software fault localization (see Fig. 2.9 for a graph presenting a comparison between different SFL algorithms performance) [Abr09].

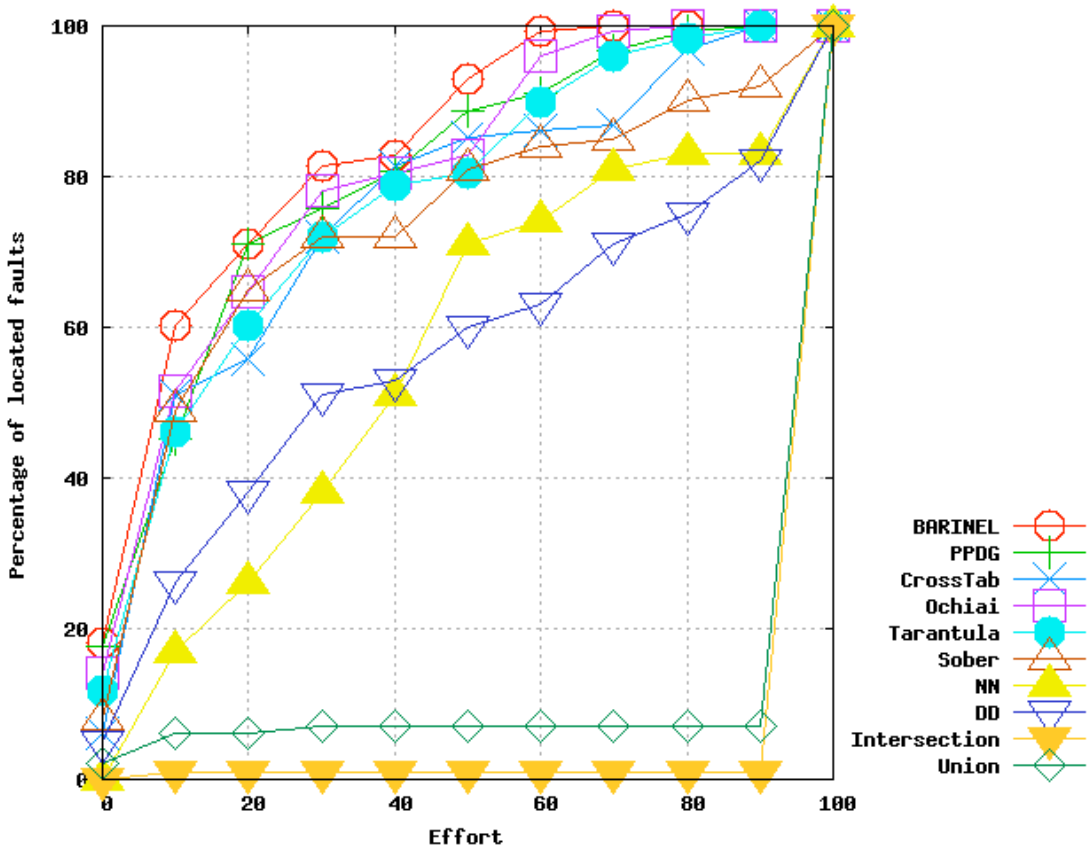

Figure 2.9: **Zoltar Algorithms Performance.** Barinel and Ochiai algorithms presents the best performance. These two algorithms are implemented on Zoltar tool.

2.3.3.2 Business Recognition

Zoltar is already being used on the industry. Zoltar tool was developed in TUDelft in collaboration with major references in global technology industries, like Philips Research Labs, Embedded Systems Institute and NXP Semiconductors. Zoltar tool has proved to be effective in real world uses, in very different situations. It was used with success in software development in different contexts, from embedded devices, like Philips TV sets, to internal software projects with considerable size.

2.3.3.3 Academic Recognition

As stated before, this tool was developed in TUDelft, and was the base project of Rui Maranhão's PhD thesis. Zoltar won the *Best Demo Award* prize at 24th Association for Computing Machinery (ACM)/Institute of Electrical and Electronics Engineers (IEEE) International Conference on Automated Software Engineering (ASE'09) [oASE11].

2.3.3.4 Active Development

Zoltar Project is being developed in the Netherlands, at TUDelft, and Portugal, at Faculty of Engineering - University of Porto (FEUP). GZoltar will be the next step of Zoltar development, and aims to make this powerful tool more usable and intuitive. Zoltar is also being developed to achieve new goals, in safety critical systems, mainly on error tolerant architectures. More than a debugging tool, Zoltar is a permanent scientific "work in progress", that aims to be a reference in the software debugging field.

2.3.3.5 Shortcomings

The main limitation of Zoltar is its graphical output. The output of the Zoltar Project is a plain text list or a source code line color highlight. Its plain text output gives a list of source code lines, ordered by the failure probability of each. It can be very hard to analyze on large projects, and is not very attractive to the programmer (see Fig. 2.10 for an example of this text-based output). This list is also difficult to navigate, because even if the information is sorted by the wanted field, it is difficult to have a clear perception about what a given line means in the entire project.

Zoltar also has a graphical implementation, to improve its output. This graphical implementation is called XZoltar. XZoltar is an enhanced version of Zoltar output. It is possible to see not only Rank and score information of each line, but also the line content, highlighted with a color that represents it's probability of being faulty (see Fig. 2.11). Although this is a big improvement over Zoltar plain text output, it does not provide a clear identification of the correlation between the system components. Is is also difficult to have a clear perception about the project structure. It is necessary to know all the project files and have a mental model about its organization.

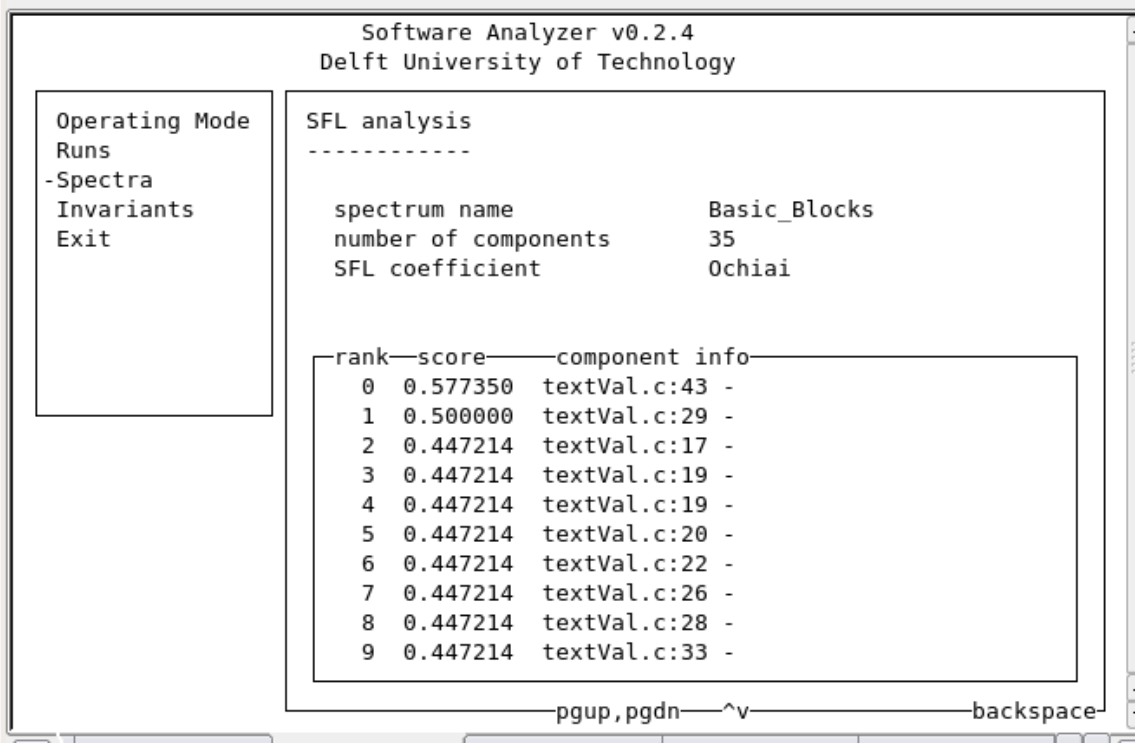

Figure 2.10: **Zoltar's Plain Text Output.** This is a text-based output, that displays a list of components and their failure probability and a rank of components, based on their failure probability.

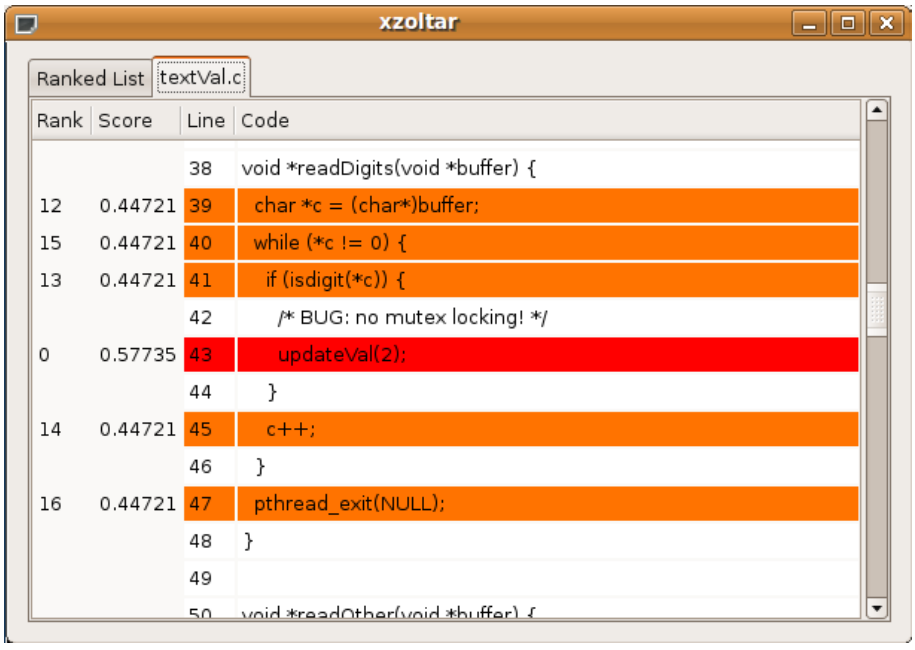

Figure 2.11: **Zoltar's Graphical Interface (XZoltar).** This interface is available for Linux, and is a viewer of source-code files, that has its lines of code highlighted with colors that represents its failure probability.

2.4 Graphical Debugging Tools Overview

The tools that were presented are currently the main references in each field. DDD is the reference in low-level graphical debugging, Tarantula is the reference in high-level graphical debugging, and Zoltar is the reference in high-level debugging due to its performance. To better understand each tools properties, it is presented a comparative table (see Table 2.1). On this table, only graphical tools are presented.

	EZUnit	**EclEmma**	**DDD**	**Tarantula**	**Zoltar**
Graphical Output	YES	YES	YES	YES	YES
Fail Probability	YES	-	-	YES	YES
IDE Integration	YES	YES	-	-	-
Hierarchical View	-	YES	-	-	-
Navigation	-	-	YES	-	-
Components Relations	YES	-	-	-	-

Table 2.1: **Current Graphical Debugging Tools Overview.** Feature comparison between presented tools.

2.5 Shortcomings

It was identified a gap in visual debugging area. There are powerful visual debugging tools, but none of them integrates with an IDE (to stay near the fault correction task), offering automatic debugging and a good visualization of the working project. The two tools that came closer to our goal was EZunit, that offered automatic debugging integrated into Eclipse (but not a powerful project view), and Tarantula, that offered automatic debugging with a better project view than EZunit, but is not integrated in any IDE. The most efficient tool in automatic debugging, Zoltar, is not integrated with any IDE and does not have also a powerful project visualization. Although these tools offers a graphical visualization of their information, they do not offer a powerful visualization of the SUT hierarchical structure, nor an easy way to navigate through all the processed information. There is a clear need for a tool that offers a powerful visualization, and integrates multiple helpful information about the SUT, to assist user in fault localization process. The best place to

a tool that helps a developer to find software faults is certainly in the same place where he correct those faults: his IDE. Currently, debugging tools that are integrated into {acIDE's do not offer an intuitive visualization nor an easy navigation of all processed data. That is the major gap that has been found.

2.6 Conclusions

Currently graphical debugging tools do not offer a powerful visualization, what compromises the comprehension of the processed information. This is even worst when developer is dealing with large projects and a lot of debugging data, that needs to be analyzed. Without a powerful visualization, that could help him to better understand all the information, and also without an easy way to navigate through that information, developer will certainly loose a large amount of time trying to understand all information processed by debugging tools. Having the most powerful tools as standalone ones, does not help him either. Having to switch between applications during debugging process (finding software fault in a tool and correct them in another), and not being able to have all the benefits of an IDE integration, such as projects and source code detection, or even interaction with the code editor, will certainly contribute to increase that loss of time during debugging process.

The idea of having a tool that could offer the best fault localization accuracy, integrated with a popular IDE, and providing a powerful project visualization was very attractive. Having identified a clear gap, and having also the needed (technological and human) resources to fill that gap, was the reason of GZoltar's birth.

At the moment, Zoltar is the most effective tool in software fault localization. It is the natural choice as a starting point for a software debugging suite. Zoltar is also very versatile, because it can be adapted to work with any SFL algorithms. This adaptability is an important factor regarding the development potential of this tool in the near future. It is necessary to process input matrix and error vector, then is also necessary to port Zoltar to Java to obtain the intended vector with the fault probability of each system component.

Chapter 3

GZoltar Project

The first goal was to develop a tool that could provide a graphical view of the project under analysis, clearly indicating the fault probability of each module. User should also be able to navigate through that graphical representation of the project, to better understand the project structure and find the faults more easily. That navigation must also report the hierarchical position of each module inside the project. In addition, there was an extra challenge: produce a light-weight way to show relations between lines of code, based on test executions analysis. These functionalities were integrated in the Eclipse IDE as a plug-in, to increase the comfort of the software developer.

3.1 Process Components

GZoltar components can be divided into five main areas (see Fig. 3.1):

- Initial Eclipse Integration;
- Zoltar Input Generation;
- Zoltar;
- Visualization Framework;
- Final Eclipse Integration;

Initial Eclipse Integration allows the detection of all open projects, classes and test classes. Zoltar Input Generation executes test cases and produce code coverage information, to create the needed SFL matrix (for further details, see section 2.1.2). Zoltar executes the Ochiai algorithm (see section 2.3.3) and processes software components relations. Visualization Framework displays different visualizations of the processed debugging data, and allows user to navigate through that information. At the end, Final Eclipse Integration creates standard Eclipse warning messages and integrates into default Eclipse code editors. For a detailed diagram about process components, see Fig. 3.4.

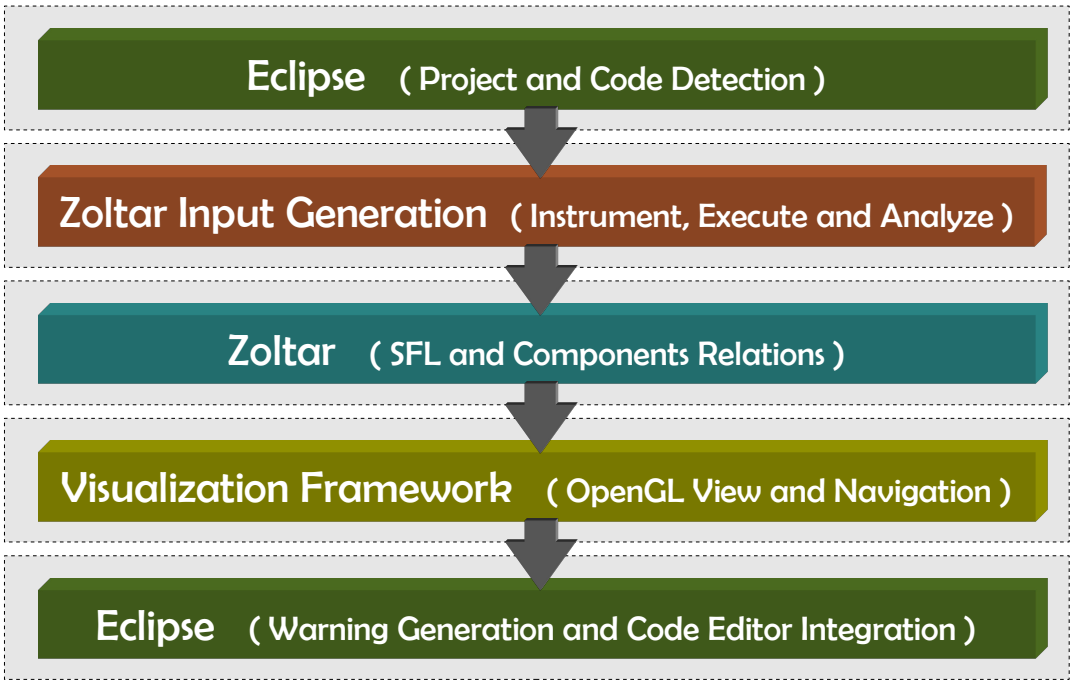

Figure 3.1: **GZoltar Brief Process Flow.** GZoltar integrates well into Eclipse. It detects its projects, processes needed data, creates a visualization on an Eclipse View and integrates with Code Editor.

3.1.1 Initial Eclipse Integration

Eclipse is a well known open source project of The Eclipse Foundation [Fou11a]. Eclipse most known application is its IDE (for more information about the way information is displayed on this IDE, see Fig.3.2). This tool is used by many programmers around the world, and is supported by many major software companies [Gee05]. It allows software development in many programming languages, even though its native programming language is Java. Because Eclipse is widely used, it is GZoltar's IDE of choice. Latest Eclipse versions (since version 3.1.2) uses Equinox. Equinox is Eclipse implementation of Open Services Gateway initiative (OSGi) R4 core framework specification [CGD99]. Eclipse plug-ins are OSGi compliant bundles [Fou11c]. An OSGi bundle can export services, and have their requirements managed by the container (Eclipse Equinox in this case). They have also their internal classpath that allow each bundle to be an independent unit [CGD99]. It is possible to automatically detect all open projects in Eclipse IDE, so GZoltar can then work with them. After having a list of the open projects, GZoltar search them for all its classes, marking test classes, those who have their name ending in "Test", to be executed later. To avoid differences between the source files and the compiled classes, at this stage it is built the entire project, to guarantee that GZoltar has the latest version of the code. This is essential to the correct work of code coverage, as explained further in more detail. After having a list of all classes from all open projects, and knowing which one are test classes, GZoltar is ready to start the test

stage.

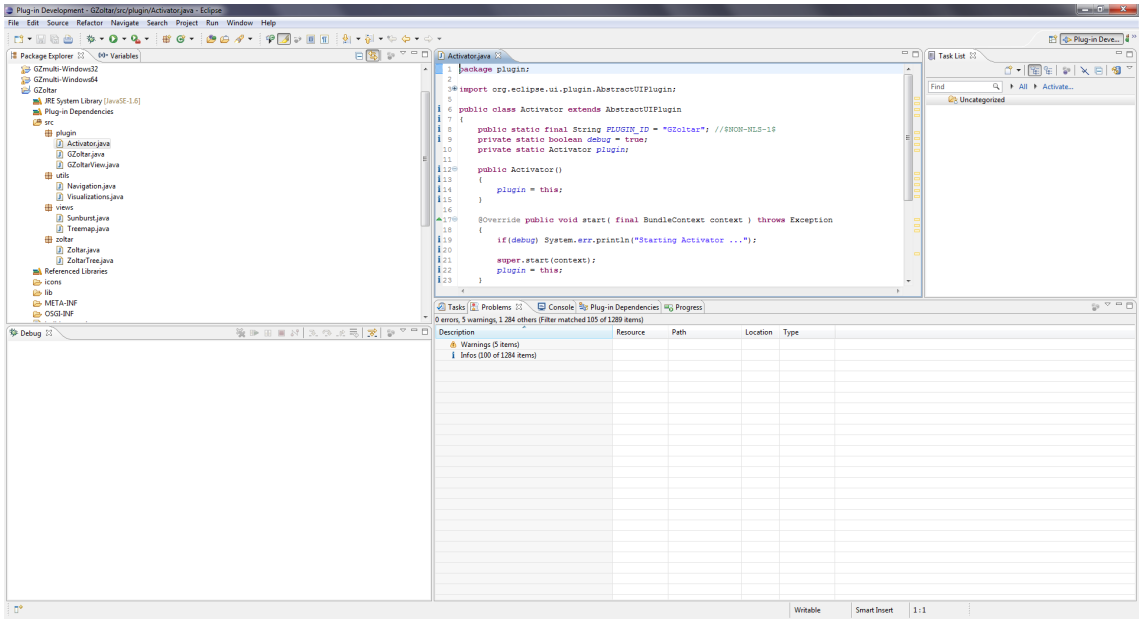

Figure 3.2: **Eclipse IDE User Interface.** Eclipse is visually composed by multiple different views. Two of the most popular ones are project explorer and code editor. There are many views that come in Eclipse default installation and others through Eclipse bundles.

3.1.2 Zoltar Input Generation

For each project there is a list of all test classes. For each test class all the code is instrumented, to allow the code coverage process. That process aims to detect if a given line of code was or not executed, during an execution. All open projects are instrumented, because open projects can call methods from other projects. After that, test classes are executed. Test classes are Java classes that implements “Callable<Boolean>”. This will allow GZoltar to call that test. It should have a “call()” method that returns a boolean value, indicating if the test passed (true) or not (false). Test classes must have a name finished in “Test”, for example: “DemoTest.java”. See Class 1 for an example of a test class.

Returned value will be stored at the error vector, to be used later by the Ochiai algorithm. Then code coverage results are analyzed. Test classes are ignored during code coverage analysis, because they are not presented on the SUT visualization. At this stage is possible to store the Zoltar Matrix line about this test (see section 2.1.2). Because the project was automatically built at the time of project and class detection, it is guaranteed that the code coverage made against compiled classes corresponds exactly to the source code files. This is essential to avoid errors in code analysis. If this procedure was not considered, code coverage information could point to the wrong line of code. What is instrumented is the compiled class, but in the Eclipse code editor is the correspondent source file that will be opened, so it should be guaranteed that they are synchronized.

Class 1 Demo GZoltar test class in Java.

```
import java.util.concurrent.Callable;

public class DemoTest implements Callable<Boolean>
{
   @Override
   public Boolean call() throws Exception
   {
      // <Insert your test code here>
      boolean testResult = (...);
      // <Return test result (boolean) at the end>
      return testResult;
   }
}
```

JaCoCo [Hof11a], a recent project from the EclEmma [Hof11b] project team was used. JaCoco can instrument Java code, execute it and analyze each line of code to check if it was executed or not. EclEmma is based on Emma [Rou11], a discontinued project. Because of that, EclEmma project team started a brand new project to create Java Code Coverage. This project is JaCoCo, which started in mid 2009. JaCoCo was been presented on Eclipse Summit Europe 2010 [Hof11c]. Java instrumentation is much more complex than a native application one, because Java code coverage tools have to deal with Java Virtual Machine (VM). JaCoCo is an Application Programming Interface (API) that allow the integration of Java code coverage functionalities on GZoltar.

Performing a code coverage analysis with JaCoCo is based on three steps:

- Instrumentation of the code to be analyzed;
- Execution of that code;
- Code coverage data analysis;

GZoltar does exactly those three steps. All code from open projects are instrumented, test classes are executed and code coverage data is analyzed. At the end, the **Zoltar!** (**Zoltar!**) matrix is built.

3.1.3 Zoltar

After collecting the input for Zoltar, it can be invoked to produce the diagnostic ranking for the observed failures. In particular, this ranking is obtained by instantiating Zoltar with the Ochai algorithm. In this tool it was used the Ochiai algorithm because it offers a good compromise between performance and results accuracy [AZvG06]. This algorithm will calculate the failure probability for each matrix component. Describing briefly the Ochiai processing, it can be said that its goal is to find the matrix column that resembles the error vector most. Each column represents a line of code, so at the end it is possible to know the failure probability of each line [JAG09].

The Ochiai algorithm calculates the failure probability considering the overall results, that means that its accuracy gets higher as the number of faults gets lower. Is is possible to clearly see that behavior when using the GZoltar tool. At the end, when only one fault is left, GZoltar accuracy is impressive. On the other hand, when the project is full of faults, GZoltar identifies areas where failure probability is higher. The same input matrix is used to process the relations between lines of code. Although initially was planned the implementation of an external code dependency graph processor, during the project development it was concluded that it was possible to use the Zoltar matrix to obtain that information. Each line represents a test execution and refers all components executed during that test. With this information it is possible to calculate their relations, by analyzing the number of parallel executions between them, and relate to the total executions of the line being compared. This results in a triangular matrix that can give us useful information about the relations between components. This information is also provided to GZoltar visualizations, as the failure probability of each component.

Because the goal is to provide an easy navigation through project structure, all the inner nodes from the project structure's tree need to be processed. That is also done, by considering the maximum failure probability value of that node's sub-nodes. At this stage all information needed for visualizations is already processed.

3.1.4 Visualization Framework

Open Graphics Library (OpenGL) is a standard specification. It defines cross-language and cross-platform API to develop 2D and 3D graphics. This technology allows us to create powerful and complex 3D scenes [KTCS05]. It is very efficient when properly used, even with a large amount of data to be represented. The main goal of OpenGL is to hide the complex interface of different 3D accelerator hardware, by offering an uniform interface. OpenGL has native libraries for each system, and can be used with C/C++ language. Nevertheless, there are bindings that allow its use in other environments, like Java. An example of one of those bindings is Java OpenGL (JOGL) [XYC05]. JOGL uses Java's Abstract Window Toolkit (AWT) [Cor11], and Eclipse uses its Standard Widget Toolkit (SWT) [Fou11b]. Fortunately, SWT has a useful class, "org.eclipse.swt.awt.SWT_AWT", that works as a bridge-builder between those two technologies. Using that class, it is possible to use OpenGL inside an Eclipse view (see section 3.2). This project was built as a multi-platform tool, to reach to a maximum number of Java developers, and to cover many development platforms. Therefore, five versions of the plugin were created: two for Microsoft Windows (32 and 64 bits), one for Mac OS X (works with 32 and 64 bit systems) and two for Linux (32 and 64 bits). This is because JOGL (OpenGL bindings) uses native system libraries. OpenGL has two main stages: initialization ("init" method) and render ("display" method).

During initialization should bo done as much pre-processing as possible, because it is executed only during the visualization startup. At this stage all needed data for the visualization is processed, and defined the common configurations. The display method is responsible for producing each image ("frame") and is executed repeatedly, so it should be highly efficient and optimized. Listeners are features that captures events. Mouse listener captures mouse clicks and movement,

and keyboard listeners captures keystrokes, for example. This information is crucial to allow user interaction. Mouse listener allows the execution of the "Picking process". That process makes the conversion between the display coordinates to the 3D scene coordinates. It also captures the object that was on a given position. This allows GZoltar to know which element was selected (see Fig. 3.3).

The visualizations are modular, and are called by the main OpenGL method. The user can select which visualization he wants to use, and that visualization is dynamically called by the visualization framework. This project allows multiple visualizations. For testing purposes, two different visualizations of tree data structures were created: a Treemap [JS91] [SCGM00] and a Sunburst [SCGM00] one. The visualization receives a tree structured data with all the SUT failure and line relations data. It also has access to necessary information by the created visualization framework, like the navigation history, component color, selected component and many other details. Visualization development gets simplified because data is pre-processed and does not change in runtime.

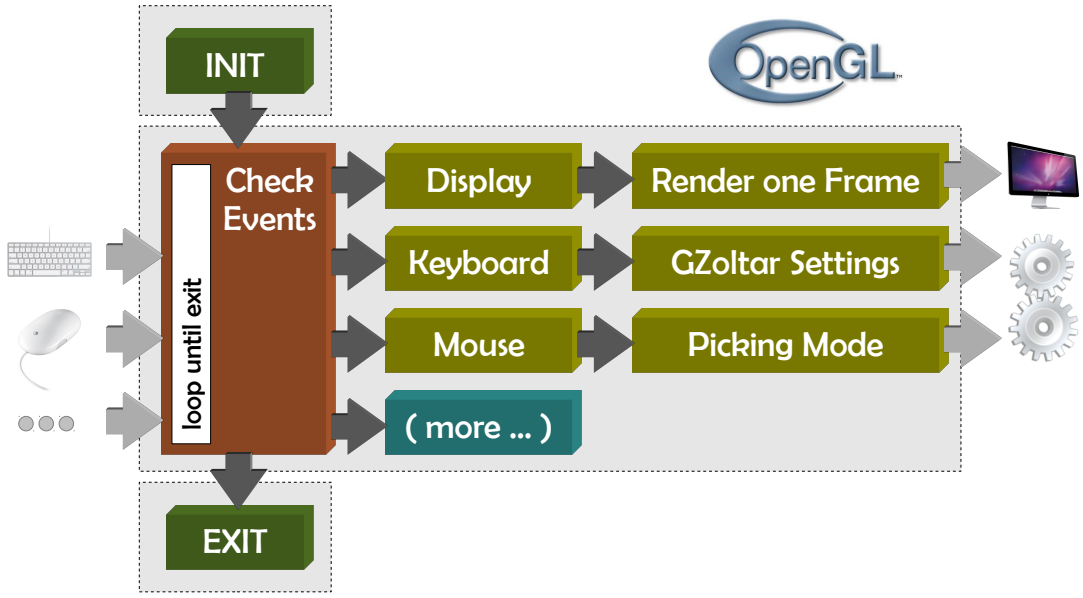

Figure 3.3: **OpenGL Cycle.** Three main stages of an OpenGL execution.

3.1.5 Final Eclipse Integration

While analyzing the SUT, user can click on the representation of a line of code, on the GZoltar visualization, and jump directly to that line. An Eclipse code editor is opened, and the text cursor placed on the selected line. There is also a list of standard Eclipse warnings, that can be consulted on Eclipse "Problems" view. Those warning messages are also displayed next to the line of code, on the code editor. Just like other standard Eclipse warnings, they are also displayed next to editor scroll bar.

With this integration, user can analyze the SUT structure, and localize its faults. He can also access directly to the code, to fix the faults in a fast way. After that, he just need to refresh GZoltar view (by pressing “F5”, see appendix B for more information), to have an updated version of the system, after the last fault fix. This way, user can perform debugging tasks (fault localization and fault fixing) at the same place, with the advantage of being a well known place (Eclipse IDE).

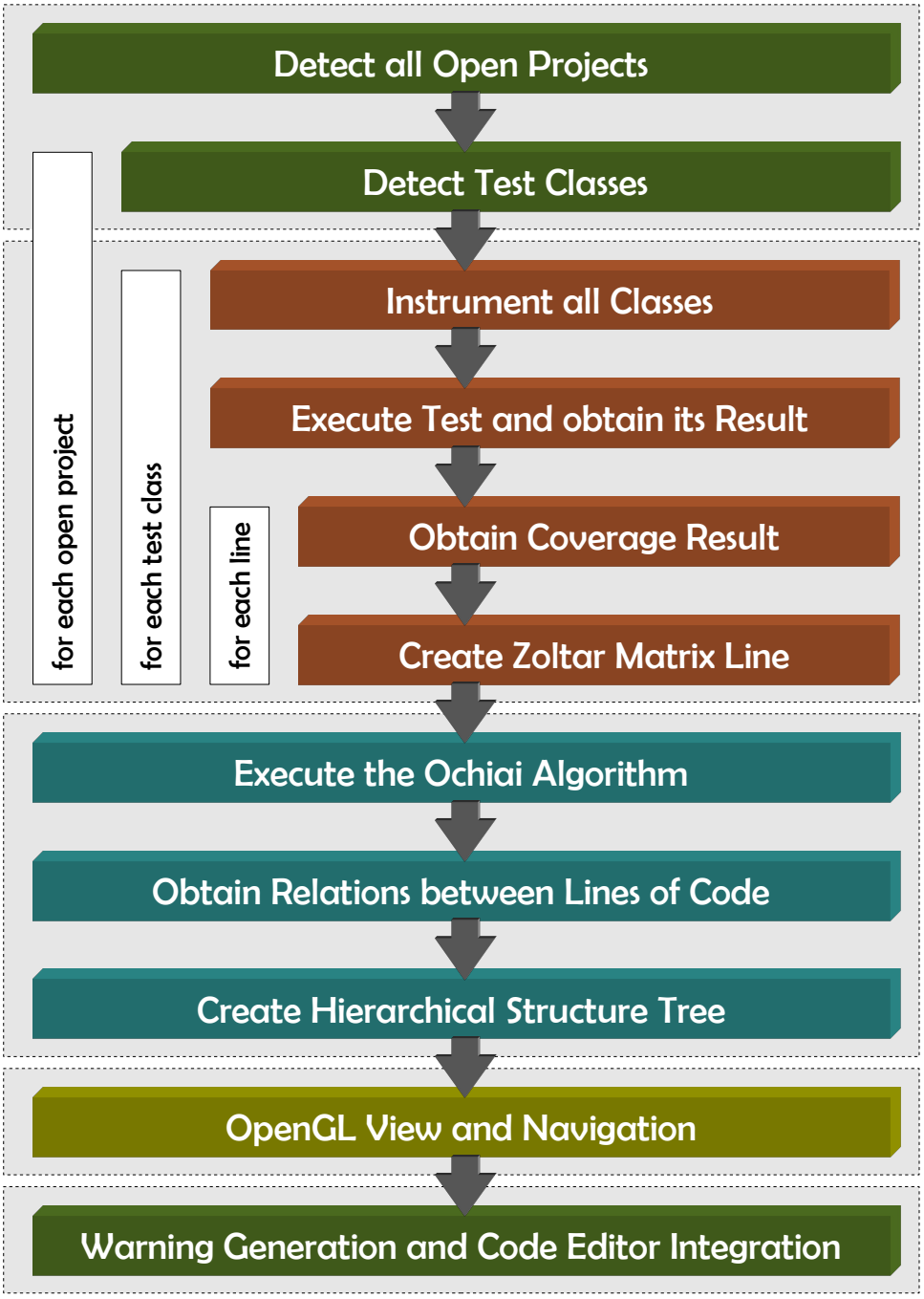

Figure 3.4: **GZoltar Detailed Process Flow.** All major tasks from GZoltar start until visualization process and code editor integration.

3.2 Logical Layers

GZoltar uses many different technologies to accomplish its tasks. It is written in Java and is an Eclipse plug-in. To obtain information about open projects, their classes, and to produce standard Eclipse warnings it uses Eclipse's Workspace component. GZoltar uses JaCoCo, a Java-based API that offers code coverage capabilities. To produce Eclipse User Interface (UI) related tasks, it uses Eclipse's Workbench component. This component has SWT, that allows the creation of GZoltar view, and has also a bridge to AWT. AWT is required because JOGL, the component that implements OpenGL bindings to Java, uses AWT to create its visualizations. JOGL is used by GZoltar to create all the 3D scenes of the different visualizations, presented on the GZoltar Eclipse view. Thus, JOGL allows GZoltar to use OpenGL technology embedded in an Eclipse View. For a schematic view of these technological layers, see Fig. 3.5.

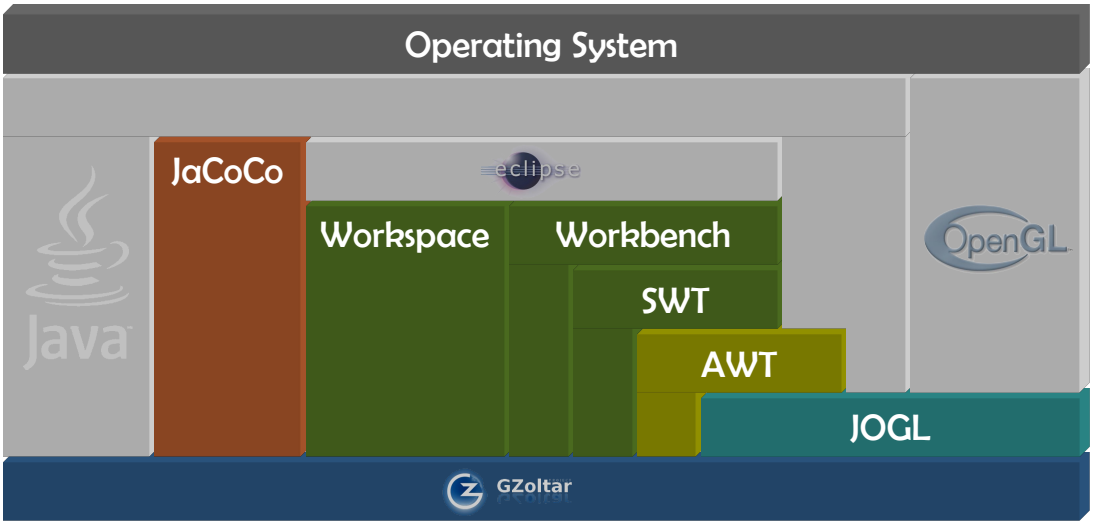

Figure 3.5: **GZoltar Layers.** Integration between GZoltar and other technologies.

3.3 Modular Architecture

GZoltar was developed in a modular way, allowing a faster and organized development in the future. It has four packages, that work much like service providers between them. The last section describes the way GZoltar code is executed, but in this section is presented the way GZoltar code is encapsulated (for a conceptual representation of GZoltar architecture, see Fig. 3.6).

3.3.1 Plugin Package

When GZoltar View starts, the class "Activator" present in this Package is called. This class has the minimum required code to startup Eclipse Plug-in. Like "Activator" class, "GZoltarView" is also a class that is executing on startup, that has the minimum required code to start GZoltar's

Eclipse View. The main class of this package is “GZoltar”. It has OpenGL core scene creation, and all needed listeners that allows user interaction. This class has all initialization procedures and also the main render class. Is the core of GZoltar. All other classes are called directly or indirectly by this one, at the startup or during render process.

3.3.2 Zoltar Package

During the OpenGL view startup, the class “Zoltar” is invoked from this package. This class has the code to process all automatic debugging results, to be used by the different visualizations. The code of this class is only executed during startup or during a view data refresh (when the user presses “F5” key. For more details see appendix B. Apart from this main “Zoltar” class, that makes all the processing, it also has a second one, “ZoltarTree” that has the model to store the processed tree structured data.

3.3.3 Utils Package

GZoltar also has a special package, called “Utils”, that has many useful services that can be used by the different visualizations. It has two classes: a “Navigation” one, that provides services related with scene navigation (zoom, pan and picking), and a “Visualizations” one, that provides services more oriented to the visualization classes, like component color processing, component label processing and other useful resources. These classes work much like service providers, that offers services to the listeners of the main “GZoltar class”, and to all visualization classes.

3.3.4 Views Package

At GZoltar’s render cycle, the selected visualization is called. Different visualizations’ code is stored at the “Views” package. Two different visualizations were created for testing purposes. This visualizations have just a minimal amount of code, because they are focused on data visualization only. They have access to the tree with all needed data, provided by the main “GZoltar” class, and to all the needed auxiliary processing is provided by the two classes in the “Utils” package. New visualizations that could be developed in the future should also be placed at this package. At any time, the user can switch GZoltar visualization by just pressing the key with their correspondent number (see Appendix B for detailed information about view switching).

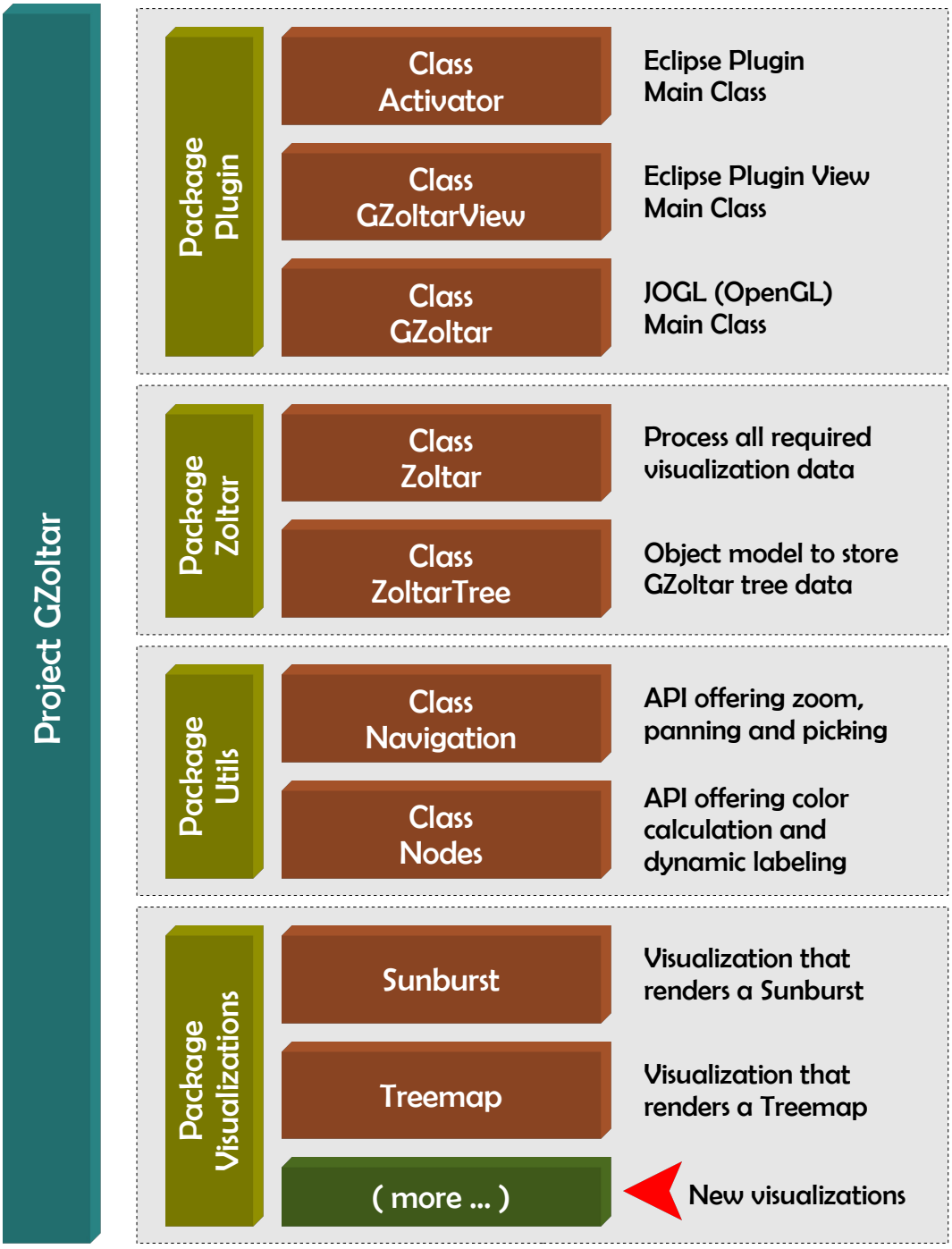

Figure 3.6: **GZoltar Modules.** GZoltar Project is compound by four packages. “Plugin” has main Eclipse bundle and OpenGL core classes, “Zoltar” has automatic debugging classes, “Utils” has OpenGL auxiliary classes, and Views has GZoltar visualizations.

3.4 Conclusions

GZoltar is able to produce accurate fault localization data, and also process relations between components. It is also able to create powerful and navigable visualizations for a project structure that uses that processed data. All this, integrated into one of the most popular IDE's. This integration provides automatic project and code detection, and Eclipse code editor opens dynamically and place it directly in the line of code that is being analyzed.

GZoltar is well integrated with Eclipse IDE, and it uses some of its internal features. Some of the features visible to the end user is the use of an Eclipse view, code editors, standard warning messages, which can be viewed on the standard Eclipse "Problems" view, and code editor tooltips, in the left side of the line of code and in the editor scrollbar. There are also some integration features that are not visible to the end user, such as project and source code detections, and automatic project building. Being multi-platform, easy to install and use (see Appendix A and B), GZoltar is ready to be used by their intended end users: Java developers.

GZoltar is also expandable, due to its modular architecture. The Ochiai algorithm can be swapped with another one in the future, without having to rewrite nothing more that a class method. New visualizations can also be created, and added without effort. This adaptability is a positive feature to allow GZoltar evolution.

Chapter 4

Case Studies

To test our automatic debugging data visualization framework, two of the most popular structure visualizations [SCGM00] were implemented : Treemap and Sunburst. However, many other visualizations can be added to this new platform.

This study aims to provide a global view of how SUT debugging data is displayed on GZoltar, to convey the system modules' failure probability and its distribution in the system.

4.1 GZoltar installation

GZoltar installs seamlessly into Eclipse. Being an Eclipse plug-in, GZoltar is installed just like any other plug-in, but have the particularity of requiring the selection of the system architecture during installation (see Fig. 4.1).

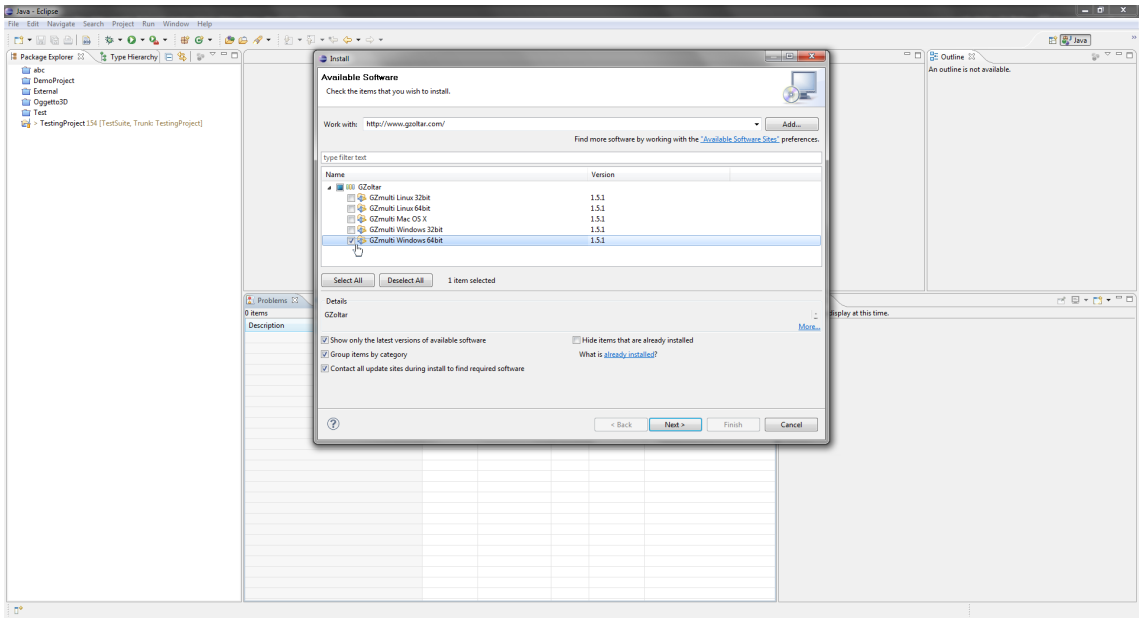

Figure 4.1: **GZoltar Installation - CPU Architecture selection.** GZoltar installs like any other Eclipse Plug-in, with the minor difference of the CPU architecture selection.

This is because GZoltar uses JOGL, which needs native OpenGL libraries to work. GZoltar is compatible with Microsoft Windows, Mac OS X and also Linux Systems (see Fig. 4.2). It is also compatible with 32 and 64 bit Central Processing Unit (CPU) architectures.

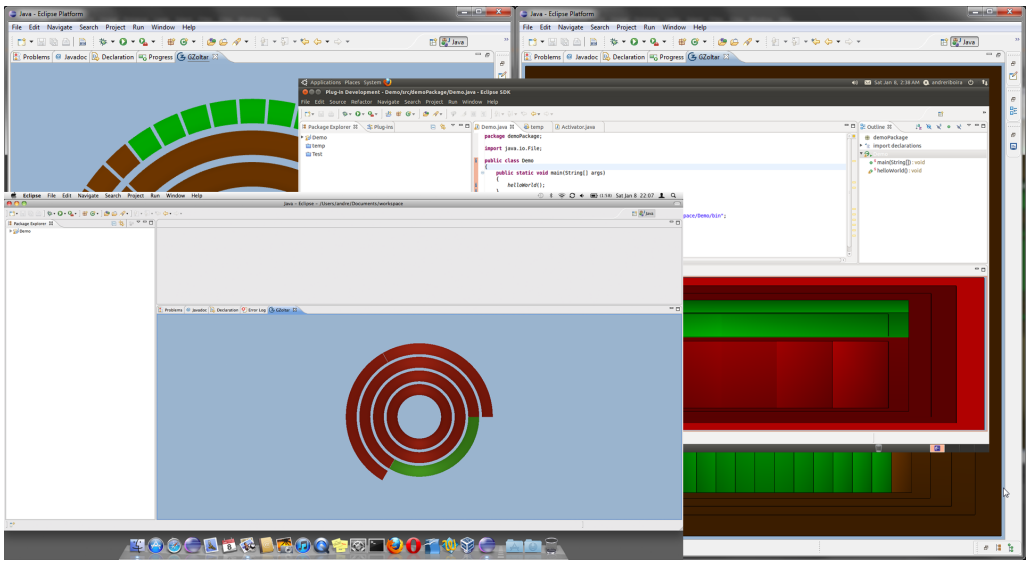

Figure 4.2: **GZoltar is Multi-Platform.** GZoltar can be installed and used without any limitation in many different systems.

Installation is fast and simple (see Appendix A), and after Eclipse's restart, GZoltar is available to use, just like any other Eclipse View (see Fig. 4.3). Every time GZoltar View is opened, all debugging data pre-processing is done. If user wants GZoltar to do that pre-processing without closing the view and opening it back again, he just have to press "F5" key (see Appendix B).

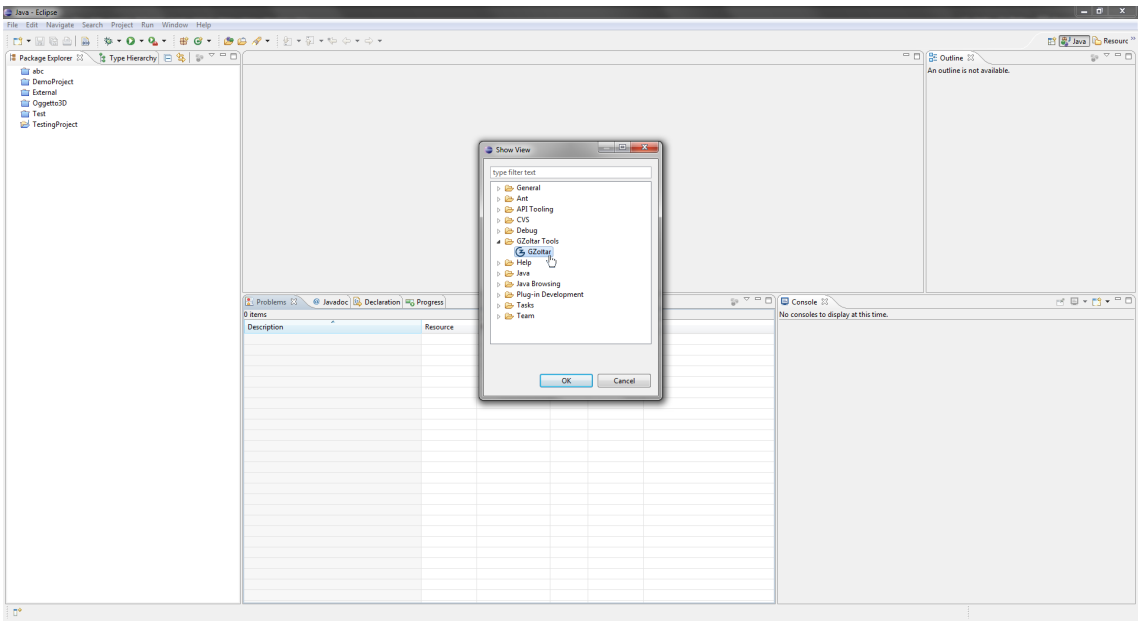

Figure 4.3: **GZoltar in Eclipse's View selection.** GZoltar works just as any other Eclipse View.

4.2 Eclipse View

By default, GZoltar presents an Eclipse View integrated into the IDE, such as many other views, like “Console” or “Problems” (see Fig. 4.4). It is possible however to expand GZoltar view by double-clicking on its view tab (with “GZoltar” label). This way GZoltar viewable area gets bigger, but all the other views gets hidden. For reader comfort, this chapter will present mainly views in full mode, to enhance GZoltar View details.

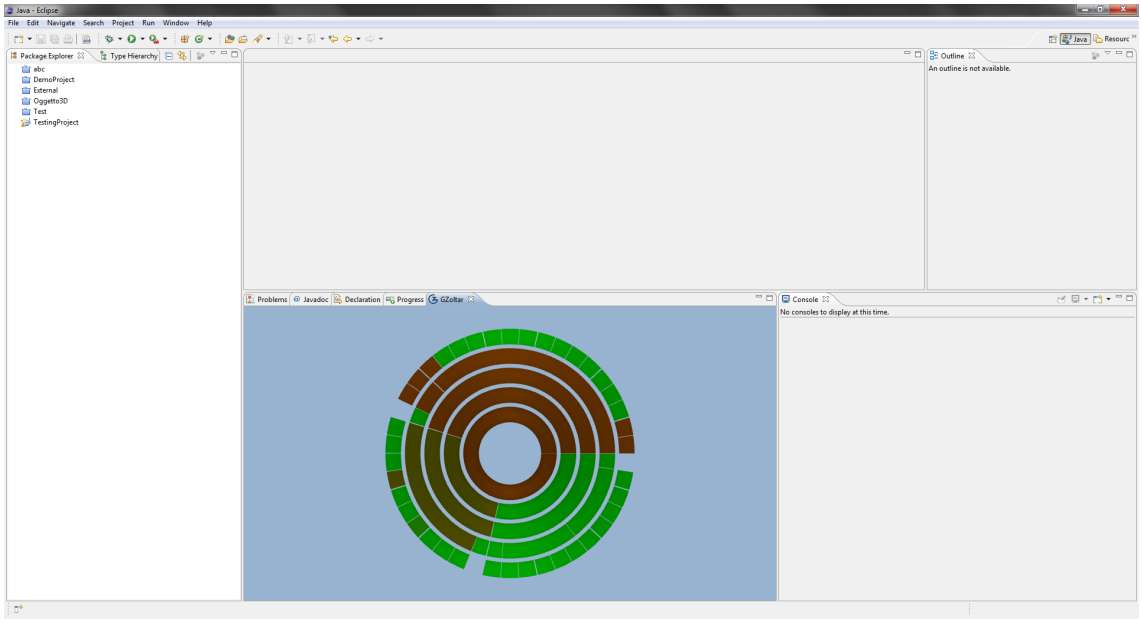

Figure 4.4: **GZoltar View in Eclipse IDE.** This is the default GZoltar View placement, however, for reader comfort, next images used in this chapter will have GZoltar View maximized.

4.3 Visualizations

To verify the way each visualization renders GZoltar data, it was created a minimal project, containing only two fictitious lines of code. The project has the following hierarchical organization:
. . . **myPro** (project)
. . . . . **org.demo** (package)
. . . . . **myClass** (class)
. . . . . **1** (line)
. . . . . **do()** (method)
. . . . . **2** (line)

This project tree has only two leafs:

- myPro/org/demo/myClass:1
- myPro/org/demo/myClass:2 [belongs to method do()]

This minimal project is intended to facilitate the understanding of data organization in the visualization. In Fig. 4.5 is displayed a comparison between Sunburst and Treemap view. GZoltar considers all packages levels, so, a composed package like “org.demo” will have two levels on GZoltar tree. This feature aims to provide a better visualization of the system structure, to help the user in its fault localization task.

Sunburst gives a view that privileges more the hierarchical location, and Treemap, on its turn, presents a view that privileges more the tree leafs. It is more difficult to understand the hierarchical position of a component in Treemap view, because blocks are overlapping by its sub-blocks, unlike in Sunburst. However, because of this characteristic, each component in Treemap have a bigger area than in Sunburst, what can be useful for selecting it. Treemap also benefits because it base area is rectangular, so it will fill the entire window area (on its default position), presenting more information in the same available space. On its turn, Sunburst by using a circular base area, have a lot of wasted space in the view.

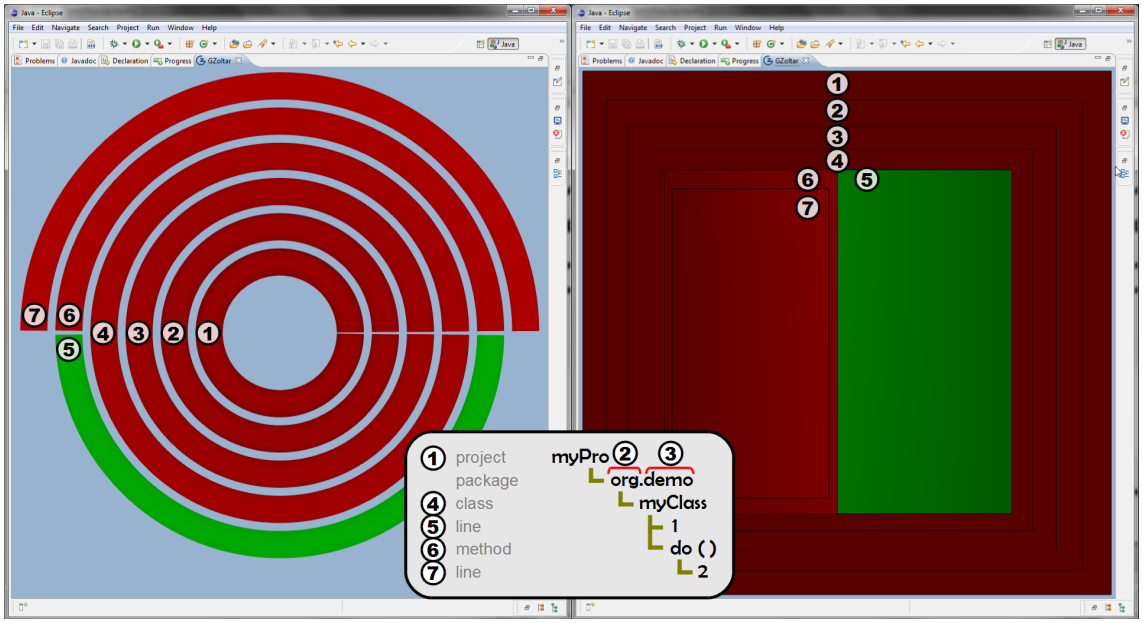

Figure 4.5: **Levels in Sunburst and Treemap.** Minimal project with levels indications.

4.3.1 Data Navigation

User can navigate through displayed data by clicking on the component he wants to expand. Expanded components shows their sub-components (see Fig. 4.6). By default, only the first tree level is opened (project level). On complex systems it is better to navigate in a step-by-step approach (by expanding only the desired sub-components) than by starting to see all the project components at once.

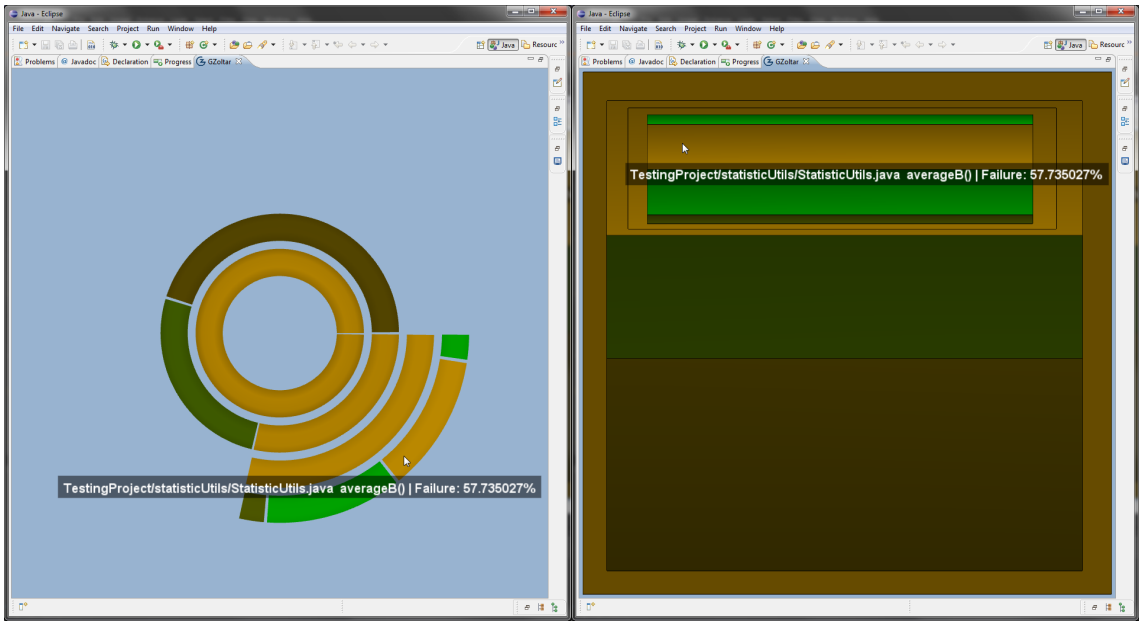

Figure 4.6: **Navigation in GZoltar visualizations.** User can expand just the needed SUT components.

If the user click on an open component, it will be collapsed. This way, user can choose what is the group of components he wants to see. When the user presses “space” key, all tree components will be displayed, the system tree visualization will be totally expanded (see Fig. 4.7). This is useful mainly on small systems, where it is possible to have a clear perception of their entire structure at once.

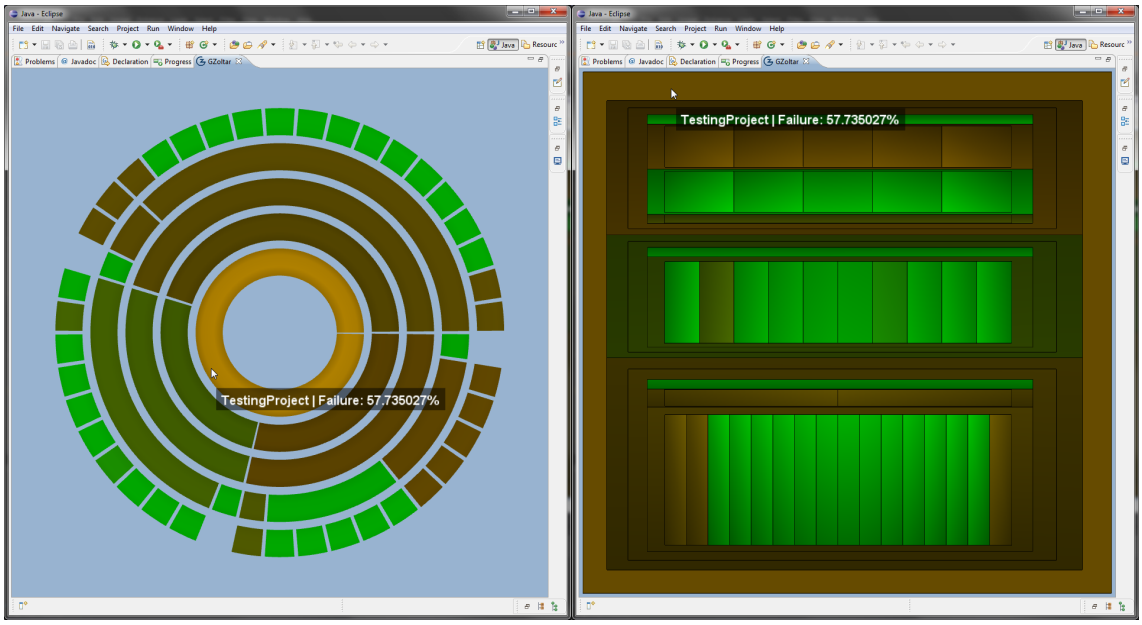

Figure 4.7: **Expanding all components in GZoltar visualizations.** User can expand all SUT components at once by pressing “space” key.

4.3.2 Zoom and Pan

This example is very short, and is easy for the user to have a quick notion about the tested system structure. Nevertheless, when zooming into a specific area of the visualization, some sense of location is lost. By zooming in, it is not possible to see all leafs, what can disturb the analysis of relations between components (see Fig. 4.8 and 4.9).

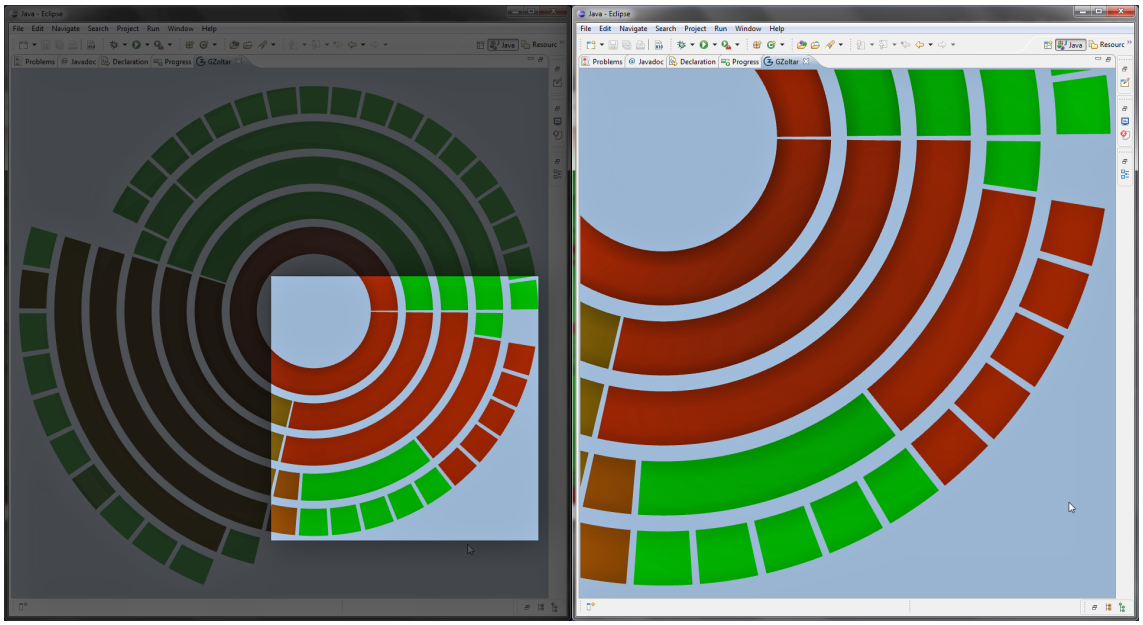

Figure 4.8: **Sunburst - Zoom and Pan on a simple project.** Detail of a project area using zoom and pan feature (right image). Left image highlights zoomed area.

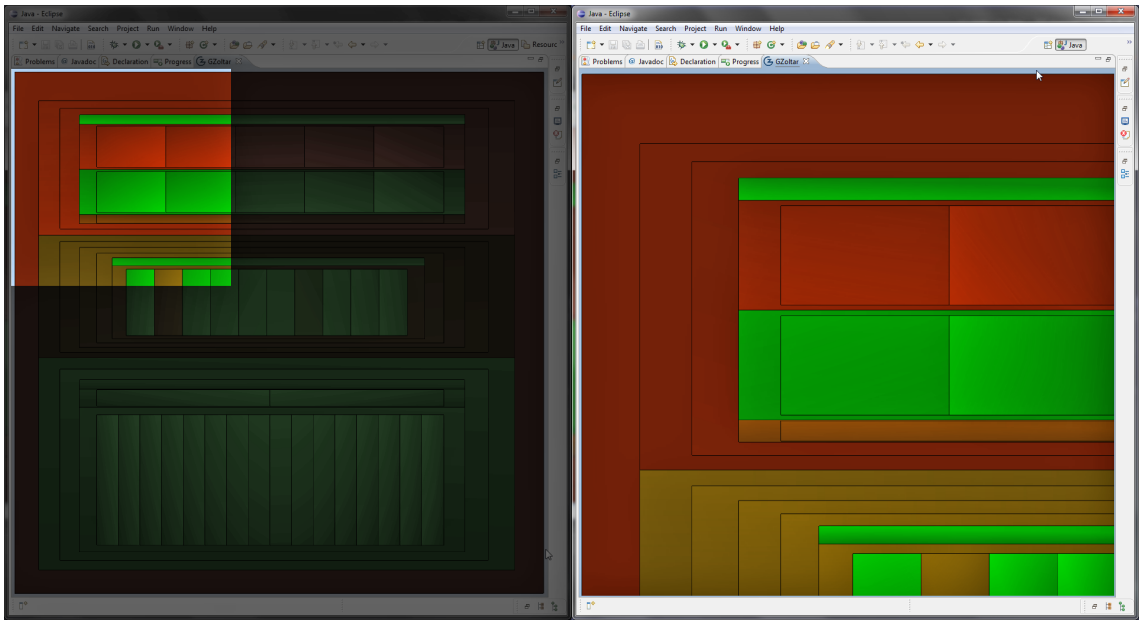

Figure 4.9: **Treemap - Zoom and Pan on a simple project.** Detail of a project area using zoom and pan feature (right image). Left image highlights zoomed area.

When to SUT is much more complex, zoom is very useful, giving much more detail in the selected area (see Fig. 4.10 and 4.11).

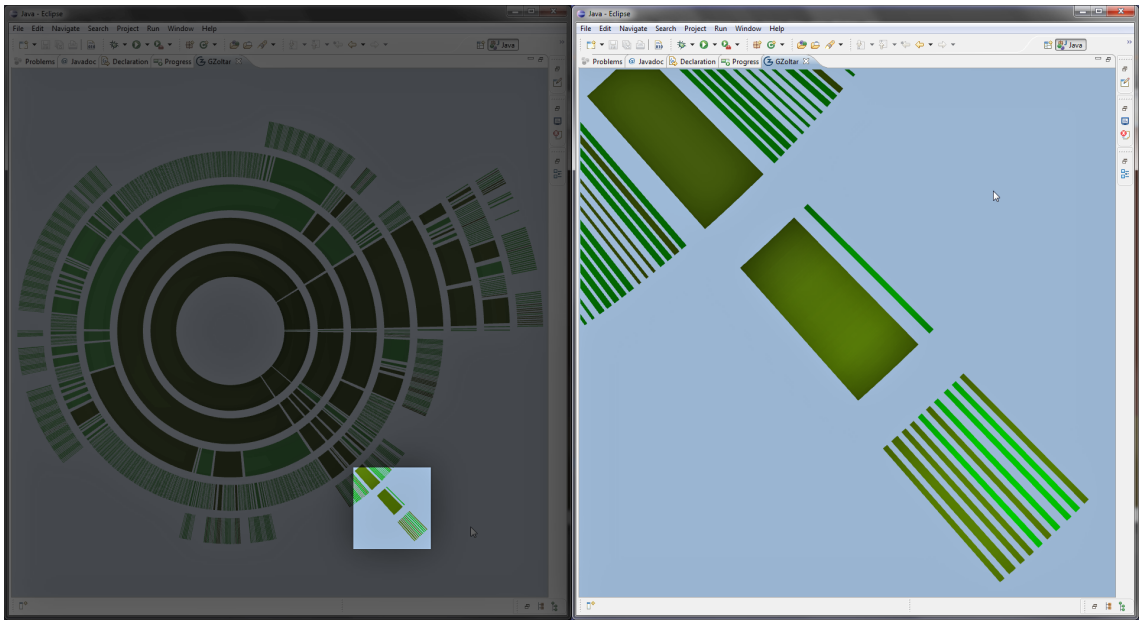

Figure 4.10: **Sunburst - Zoom and Pan on a complex project.** Detail of a project area using zoom and pan feature (right image). Left image highlights zoomed area.

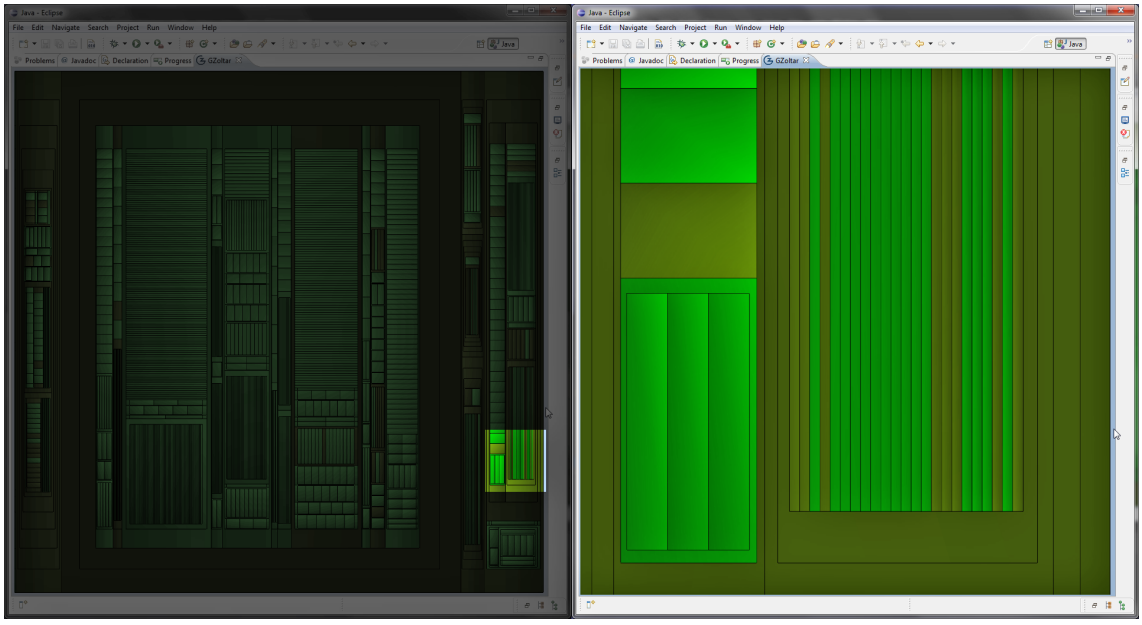

Figure 4.11: **Treemap - Zoom and Pan on a complex project.** Detail of a project area using zoom and pan feature (right image). Left image highlights zoomed area.

4.3.3 Root Change

Root change can be seen as a “smart zoom”, because the viewing area gets limited to increase visualization detail, but maintaining the same visual structure concept. Root change has a limitation comparing to zoom: if user want to navigate to a lower level, he has to make another root change, but in zoom mode he can not only zoom out but also pan. With this ability, he can explore adjacent components quicker than making successive root changes. This can be verified in both visualizations, Sunburst and Treemap (see Fig. 4.12 and 4.13).

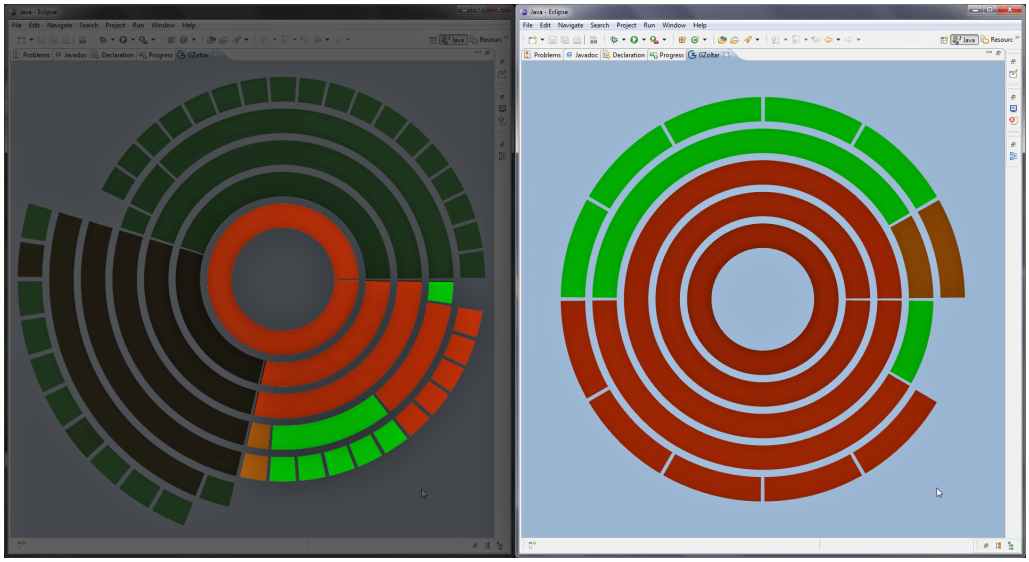

Figure 4.12: **Sunburst - Root Change on a simple project.** Detail of a project area using root change feature (right image). Left image highlights affected area.

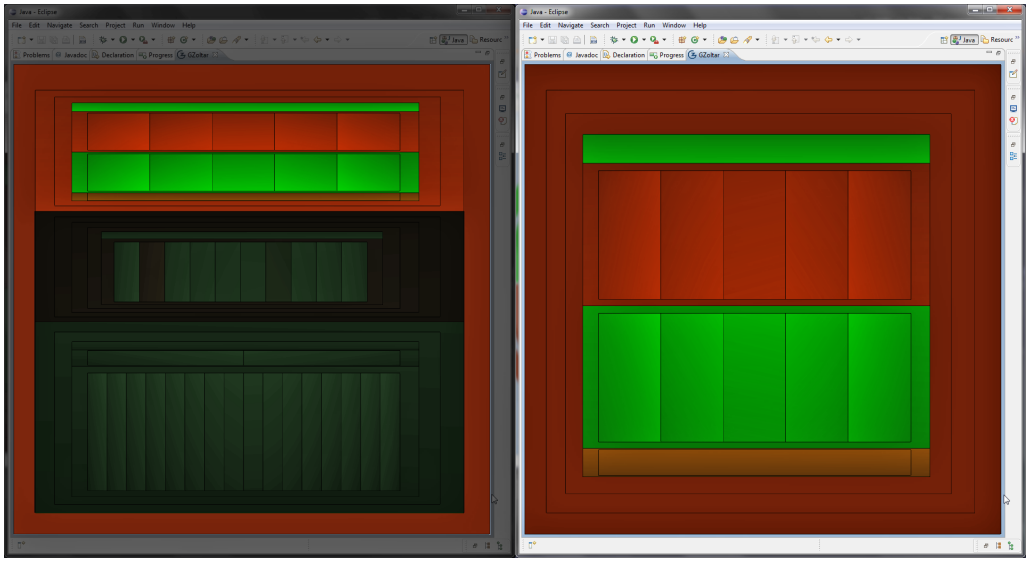

Figure 4.13: **Treemap - Root Change on a simple project.** Detail of a project area using root change feature (right image). Left image highlights affected area.

A root change is a useful feature to analyze big projects. It is possible to render successive visualizations having different components as root. Like this, it is possible to analyze some part of the SUT with maximum detail. Falling into a conceptual problem is obvious: It is hard to see a big system representation at once, with a high detail level, because there are many limitations, including the display resolution. It has to be found some workaround to this problem, by finding ways to reach a good balance between detail amount and the displayed area percentage of the whole system (see Fig. 4.14 and 4.15).

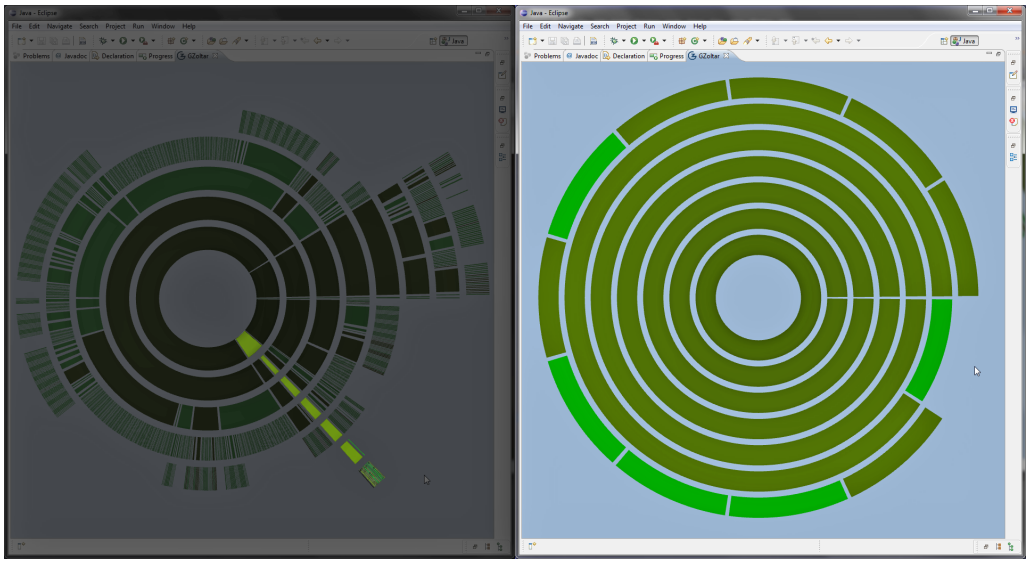

Figure 4.14: **Sunburst - Root Change on a complex project.** Detail of a project area using root change feature (right image). Left image highlights affected area.

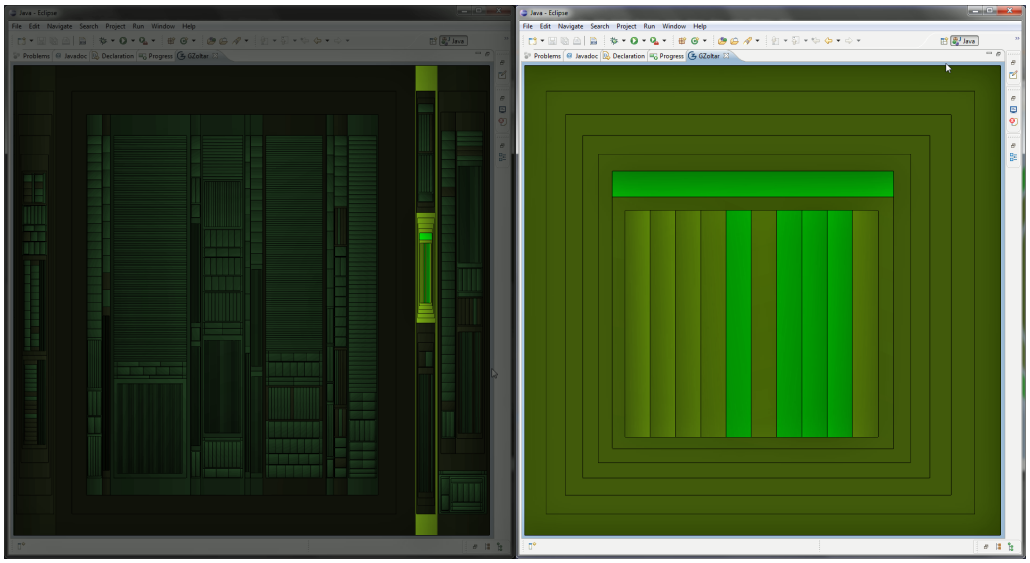

Figure 4.15: **Treemap - Root Change on a complex project.** Detail of a project area using root change feature (right image). Left image highlights affected area.

4.3.4 Code Coloring

It is possible to see if a project has faults by analyzing its components colors. If a project does not fail in any test, all components will be rendered in green, otherwise, a color that varies from green to red will represent the component failure probability. It is possible to place the mouse over any project component, at any level, and see its name and failure probability value (see Fig. 4.6). All this information corresponds to the real project file structure.

In the last level, when working with Sunburst visualization, the component color represents not the individual failure probability but the relations between components. The color of each leaf node will vary from the one that the selected component has to gray, revealing the amount of parallel executions with the selected component. It is possible to have a notion about the way components relate with each other. With this feature it is possible not only to know if components are related but also how deep is that relation. Treemap did not implemented this feature because it is a leaf-centered view. The tree leafs have the majority of space in the visualization, so the user would difficultly see the component with individual failure probability coloring (see Fig. 4.16 and 4.17).

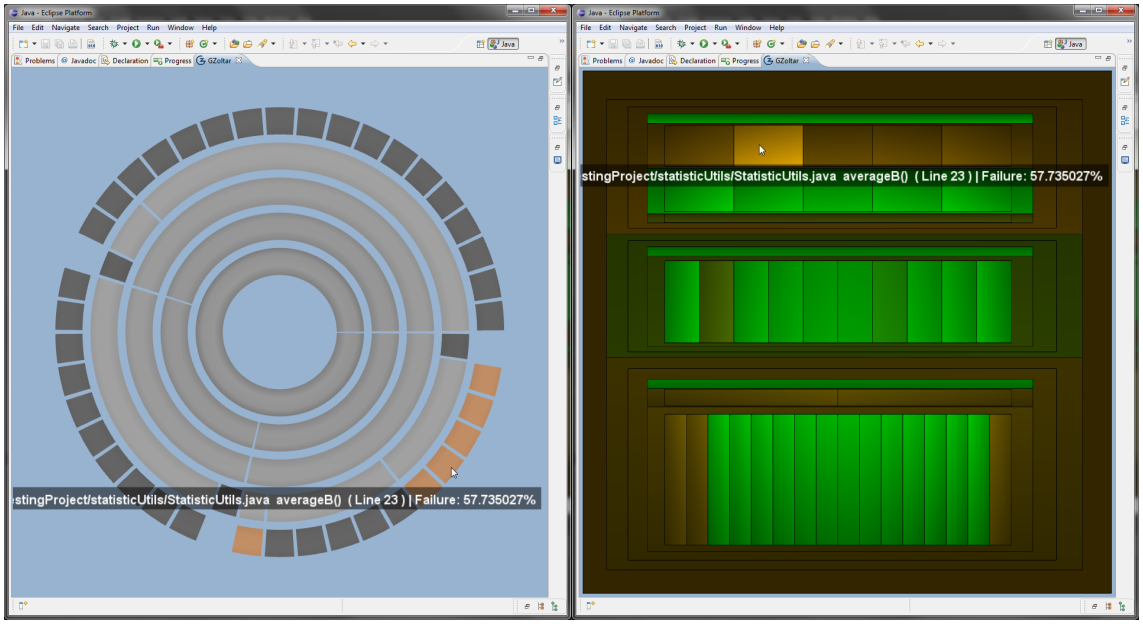

Figure 4.16: **Sunburst and Treemap with a Demo Project - Leafs.** In this example is possible to observe that there is another component that is executed every time the selected component is, because it has the same color.

Result accuracy is not related directly with GZoltar, but with the SFL algorithm that is implemented: Ochiai. Being modular, GZoltar can easily be updated to incorporate any other SFL algorithms in the future, if there is any benefit on that. User should have in mind that GZoltar is intended to be only a visualization tool and should not be seen as a closed system, but as an expandable and adaptive one. When the Ochiai algorithm presents an high fault localization accuracy, i.e. when the system has few faults, color differences are so high between good and faulty

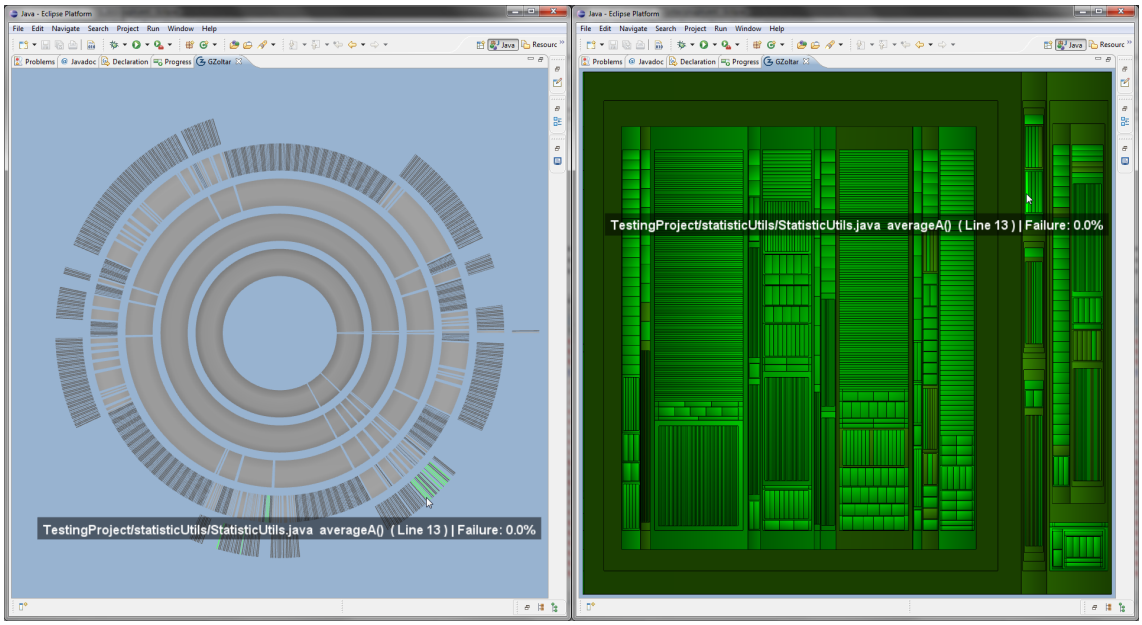

Figure 4.17: **Sunburst and Treemap with Multi Projects - Leafs.** In bigger projects is still possible to analyze relations between lines, although not being as easy as in smaller projects.

components, that visualizations in these cases works like if the faulty component activated an alarm, because it is really obvious (see Fig. 4.18).

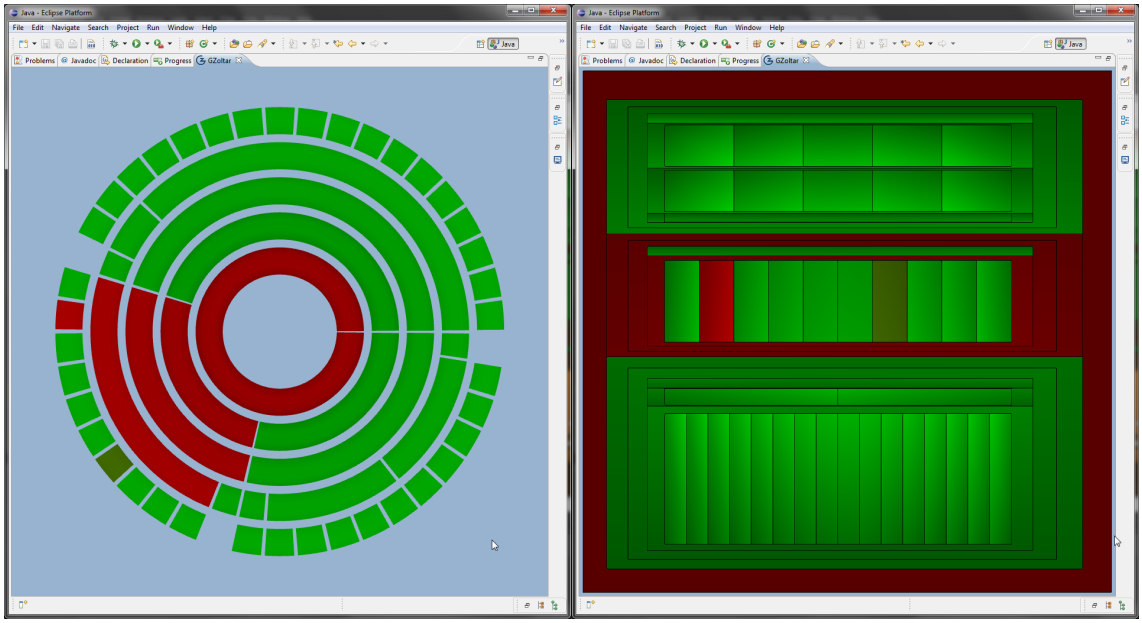

Figure 4.18: **Sunburst and Treemap with a Demo Project - Different colors.** When the Ochiai algorithm has a very high precision on its results, the localization of the faulty component is almost immediate.

On its turn, when failure probability is very similar, color gets obviously also very similar. On these cases, is hard to figure out which component has the higher failure probability (see Fig. 4.19). Fortunately, this only happens when the Ochiai algorithm is not so accurate, i.e. when user is dealing with projects with many disperse errors.

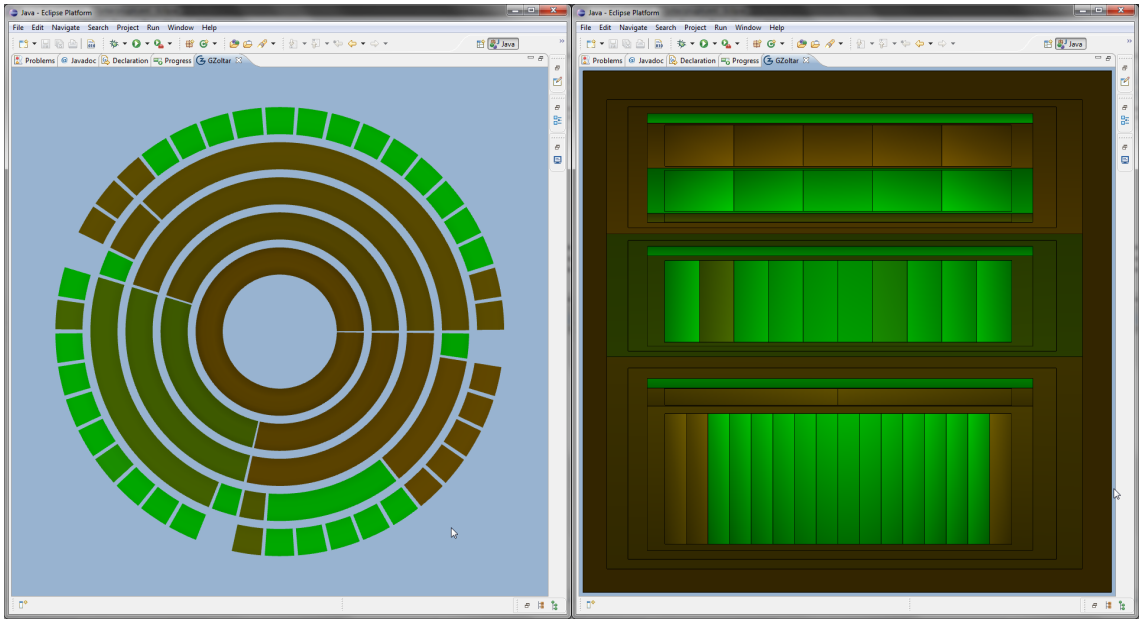

Figure 4.19: **Sunburst and Treemap with a Demo Project - Similar colors.** When the Ochiai algorithm accuracy is not that hight, it could be harder to localize the faulty component, mainly because on these cases there are more than one fault on the system that has been executed during tests.

4.4 IDE Integration

GZoltar integrates well with Eclipse features. GZoltar can open directly the standard Eclipse code editor, and position the text cursor on the wanted line (see Fig. 4.20). It also processes standard Eclipse warnings, so they are available immediately after the line of code (see Fig. 4.21), and also next to editor scrollbar (see Fig. 4.22). Those warnings are also available at standard Eclipse's "Problems" View (see Fig. 4.23), and user can also jump to the respective class and line of code by double clicking on the warning message he wants.

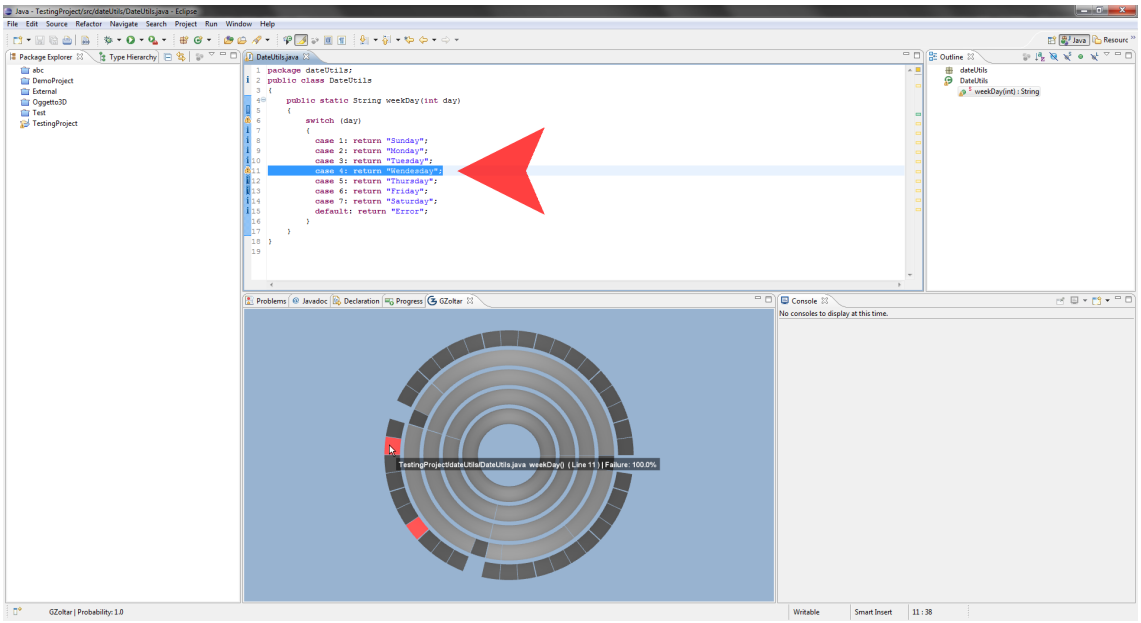

Figure 4.20: **Code Editor Integration.** When user clicks on a line of code representation, the corresponded code editor is automatically opened.

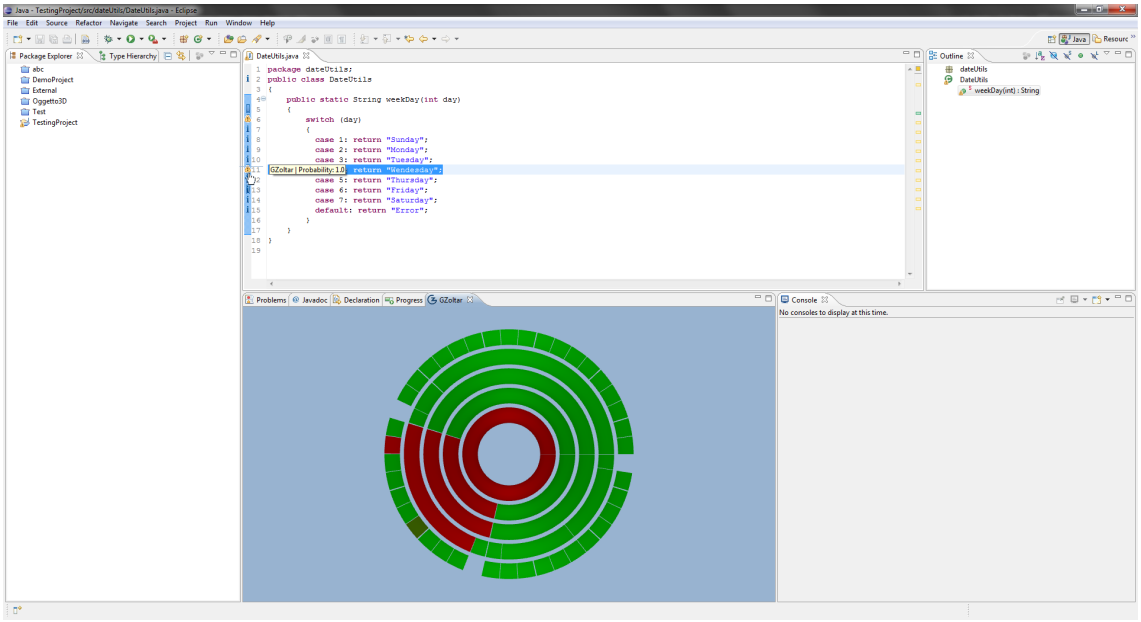

Figure 4.21: **Warning tooltips next to line of code.** Immediately after the line of code, user finds signs that reveals if that line has or not any failure probability. Those signs have tooltips with the failure probability value.

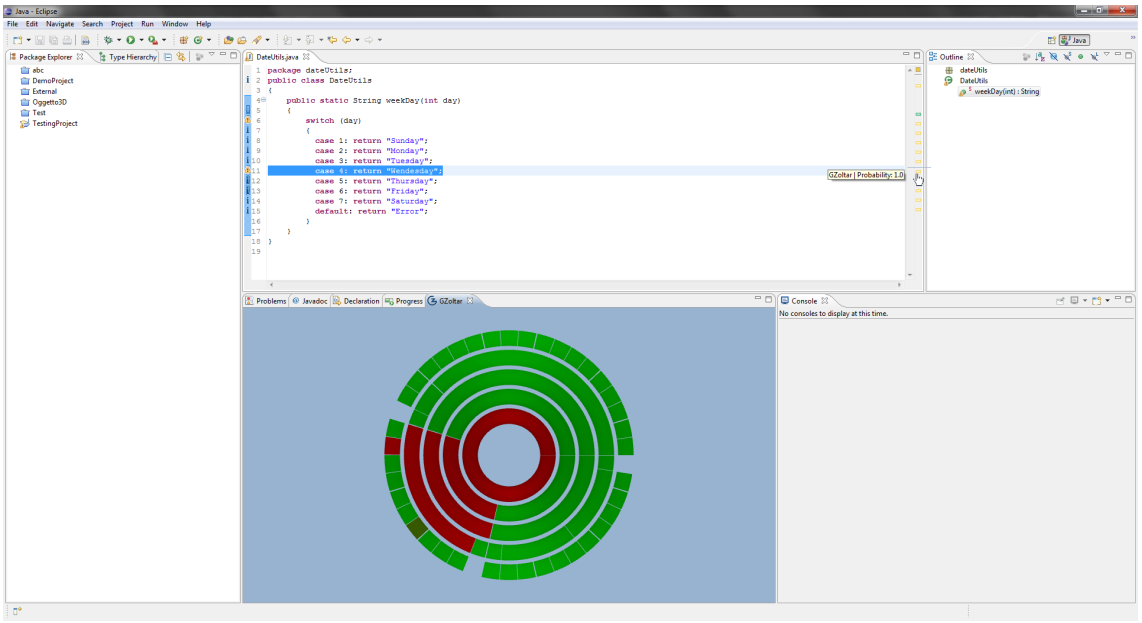

Figure 4.22: **Warning tooltips next to scrollbar.** Next to scrollbar user finds traditional Eclipse warning tooltips, showing GZoltar failure probability of that line of code.

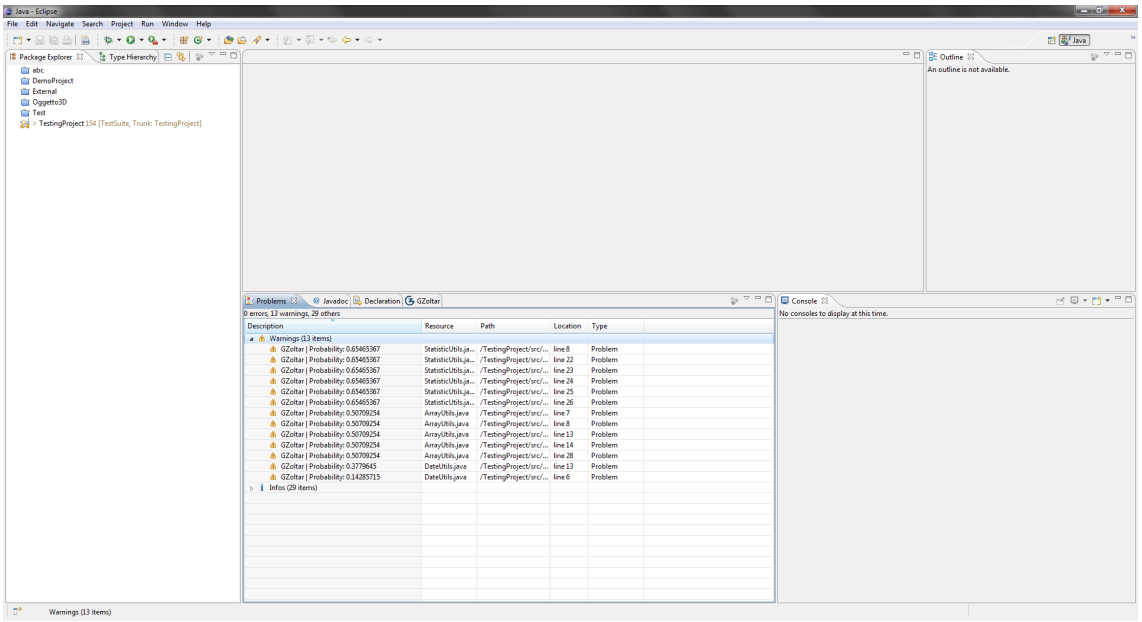

Figure 4.23: **Problems View.** Because GZoltar uses standard Eclipse warnings, they are also shown on Eclipse “Problems” View.

4.5 Conclusions

It can be concluded that GZoltar Visualization Framework is working as expected. The two visualizations (Sunburst and Treemap) rendered perfectly all input data and navigation is intuitive. Its performance is very acceptable, mainly because GZoltar Visualizations core uses OpenGL. A developer can have a fast glimpse of the SUT hierarchical structure, and also of its failure probability. He can also navigate through any of the available visualizations, and analyze in detail a sub-part of that system. He can also analyze the way system components correlate with each other. By having all this integrated in Eclipse IDE, that allows interaction with code editors and default warnings, certainly the debugging task will be enhanced and more productive. By comparing this GZoltar with other available graphical debugging tools, it can be concluded that GZoltar fills the gap that existed in this area. See Table 4.1 for a comparison between GZoltar and other tools.

	EZUnit	**EclEmma**	**DDD**	**Tarantula**	**Zoltar**	**GZoltar**
Graphical Output	YES	YES	YES	YES	YES	YES
Fail Probability	YES	-	-	YES	YES	YES
IDE Integration	YES	YES	-	-	-	YES
Hierarchical View	-	YES	-	-	-	YES
Navigation	-	-	YES	-	-	YES
Components Relations	YES	-	-	-	-	YES

Table 4.1: **GZoltar Comparing to Other Current Graphical Debugging Tools.** GZoltar fills the gap of other graphical debugging tools.

Chapter 5

Conclusions and Future Work

At the moment, there are some basic debugging tools integrated on IDE's, and none of them offers a powerful project structure visualization. Besides, there are also some powerful manual debugging tools external to the IDE's, like the well known DDD [ZL96]. DDD is probably the reference in visual debugging, although it requires heavy user interaction. On the field of automated debugging, the tool with the most powerful interface is certainly Tarantula [JHS02b], but their results are not as effective as Zoltar's [AZGvG09]. Zoltar, in turn, has the best performance in automatic debugging, but it lacks an intuitive user interface.

5.1 State-of-the-art of Automated Debugging Tools

Zoltar is currently the tool that has the best performance among all the others. Its performance was already proved to be better in some scientific studies and in experiments on the field. This is because Zoltar uses two of the most effective algorithms of software fault localization: Barinel and Ochiai [AZGvG09]. Zoltar is also a versatile tool, and can implement other SFL algorithms, so it can easily evolve in the future, even if better algorithms of the same type are developed with a better performance than this ones.

Although Zoltar does its work in a very effective way, it lacks an important feature: a good interface. This is an important feature because it will actuate directly on the delay that the user will have from executing the tool and understand and use its results. This is important, and should be taken into consideration. All efforts should be gathered to make the application faster, and speed does not end at software output. Its understanding should also be considered.

5.2 GZoltar Project

Initially, GZoltar project goal was only the creation of an advanced but standalone visualization tool, to work directly with Zoltar original tool, directed to C/C++ projects.

Shortly thereafter, those goals become much more ambitious. The idea was to integrate Zoltar into a popular IDE. Because of its popularity, development versatility (due to its plug-in architecture), and because it is an open source project, Eclipse was the chosen IDE. It was the first obstacle, because that required a port of Zoltar to Java, and find a way to process all its needed input. JaCoCo was the solution for the input Zoltar matrix. It was also necessary to have a way to obtain the test results, that were executed in a different environment from the one used by GZoltar plug-in. To solve that issue, it was defined that test classes should implement the "Callable<Boolean>" interface, so that all test classes could be called, and their result could easily be obtained to proceed the Zoltar input matrix building.

There was also the idea of implement a way to process relations between lines of code, to further assist the developer in fault localization. After search for an external API to do that task, it was concluded that zoltar input matrix already had all required data, obtained by the code coverage process. It was then developed a routine to process that data from the input matrix, to be used by the visualizations.

At the beginning, only one visualization was considered. After some research, it was also concluded that there were no tree structure visualization concepts that fits well in all situations. Some are best for one kind of tree structure, and others are best for other kinds of trees. Having that, the implementation of more than one visualization was obvious. It was chosen Treemap and Sunburst concepts because their popularity and because they somewhat complement each other. The first is more leaf centric and the last is more hierarchical structure centric. Although these are two powerful visualization concepts, GZoltar was developed having in mind that it could have many more visualization concepts in the future.

Another concern was that GZoltar should be multi-platform, to cover a wider group of developers. That represented some difficulties, because GZoltar visualizations uses OpenGL, and although OpenGL is multi-platform, it has native system libraries, for each different system. That was an obstacle to distribute GZoltar as a standard Eclipse plug-in. It was necessary to proceed to the creation of five different GZoltar versions, because of that issue.

5.3 Knowledge Transfer

Zoltar did succeeded in academic and industrial field, as already mentioned. It would be great if GZoltar follow the same route. GZoltar already aroused the interest of international academic and industrial community, started by a presentation of its concepts at International Conference on Testing: Academic & Industrial Conference Practice and Research Techniques (TAIC-PART) 2010 [TAI10]. Certainly, GZoltar will be presented in future conferences of related fields. MAP Doctoral Program in Computer Science (MAPi) [iCS11] students are also interested in GZoltar. André Riboira, GZoltar main developer, lectured a class presenting GZoltar to the current MAP-i PhD students, and got positive feedback.

Being the most effective tool in software fault localization, Zoltar aroused interest from several institutions. GZoltar as "the next version of Zoltar" will certainly follow the same route. It is a

great responsibility to develop such a tool, that should be ready to be used for important institutions like the ones that have expressed their interest on Zoltar project. One of those institutions is Strongstep - Innovation in Software Quality. Strongstep have the important task of promoting the enhancement of software development processes of their clients. Strongstep is partner of Software Engineering Institute (SEI) and European Software Institute (ESI), and has as clients important companies and national government departments. Their team is specialized in Capability Maturity Model Integration (CMMI), International Software Testing Qualifications Board (ISTQB), Team Software Process / Personal Software Process (TSP/PSP), Project Management Book of Knowledge (PMBOK), Information Technology Infrastructure Library (ITIL), Rational Unified Process (RUP), Scrum and Extreme Programming (XP), among other methodologies and technologies.

Virtually any Java Programmer can become a GZoltar user. Java as chosen mainly because it is a very portable technology, running on different operative systems and even on different devices, not only personal computers but also the embedded ones. Although GZoltar can be used by anyone, with certainly good results, GZoltar should arouse more interest in all safety critical system developers. Anyone that takes its system safety seriously, and invests a lot of resources in software testing and fault corrections, will certainly find GZoltar as a powerful tool, because it uses Zoltar, the most effective tool in software fault localization. Having an intuitive interface integrated into a popular IDE will enhance software faults researchers productivity, because they will be able to see clearly the faults localization, and will also have sharper view of all the project structure.

Finding software faults faster will allow software testers to find more faults in the same amount of time. This will lead to a higher software reliability level, or to a decrease of the software test period, which may mean a significant cost reduction.

5.4 Future Work

Although the initial goals were all achieved, even much more new goals that came during GZoltar development were achieved. There are always new ideas, on an almost daily basis, that could improve GZoltar project. Even during the case studies, when GZoltar plugin was considered ready, there were new good ideas to improve GZoltar behavior.

5.4.1 Mini Map on Zoom In

One idea for future work is to implement one of the widely used feature in video-games industry: a mini-map [WF11]. This would certainly be a big advantage to have a higher level of detail without loosing the sense of location in the entire system. However, this feature seems only useful in zoom and pan mode. In root change cases some changes have to be done to this concept, for instance presenting a full view of the entire system (in a mini-map way), but having the components that are being represented by the new visualization somehow highlighted, and not with a rectangle representing the viewing area, like in traditional mini-maps.

5.4.2 Spectrum Color Bar

Another idea is that it could be useful to have a reference bar, with a spectrum from red to green, so the user could compare at any time the selected component color with the position of that color in the reference bar. This way user could easily understand the meaning of that color, in terms of failure probability.

5.4.3 Testing with Humans

It is important to test this kind of tools with their target. It is planed the performance of tests with Java software developers, where GZoltar will be tested against other alternatives. This also allow the measurement of the effectiveness of GZoltar.

5.4.4 JUnit Integration

JUnit is an unitary test framework for Java [Lou05]. JUnit is widely used to test Java projects, so it would be very interesting to integrate it with GZoltar, to be an input source to create Zoltar input matrix.

5.4.5 GPGPU

Ochiai algorithm is parallelizable. That should also be explored, because the use of General-Purpose computation on Graphics Processing Units (GPGPU) [WL08] could bring a performance boost to GZoltar, in respect of the Ochiai algorithm.

5.4.6 New Interaction Paradigms

New interaction paradigms are appearing frequently, and some of them are getting a massive acceptance, like multi-touch devices. This capabilities could help the interaction between users and GZoltar, so they should be explored.

5.4.7 New Visualizations

Although Treemap and Sunburst are two powerful visualizations, new visualizations should be implemented in future. Not all users like the same visualizations, so it would be interesting to have many more visualizations, to achieve a larger number of happy users.

References

[Abr09] Rui Abreu. *Spectrum-based Fault Localization in Embedded Software*. PhD in computer science, Delft University of Technology, 2009.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004.

[Avi71] A. Avizienis. Faulty-tolerant computing: An overview. *Computer*, 4(1):5 –8, 1971.

[AZGvG09] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009.

[AZvG06] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *Dependable Computing, 2006. PRDC '06. 12th Pacific Rim International Symposium on*, pages 39 –46, 2006.

[BH08] J. Börcsök and P. Holub. Basic consideration for sil calculation in safety systems. In *ACACOS'08: Proceedings of the 7th WSEAS International Conference on Applied Computer and Applied Computational Science*, pages 219–227, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[BWK07] Stefan Berner, Roland Weber, and Rudolf K. Keller. Enhancing software testing by judicious use of code coverage information. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 612–620, Washington, DC, USA, 2007. IEEE Computer Society.

[CGD99] M. Condry, U. Gall, and P. Delisle. Open service gateway architecture overview. In *Industrial Electronics Society, 1999. IECON '99 Proceedings. The 25th Annual Conference of the IEEE*, volume 2, pages 735 –742 vol.2, 1999.

[Coh94] I.B. Cohen. The use of "bug" in computing. *Annals of the History of Computing, IEEE*, 16(2):54 –55, 1994.

[Cor11] Oracle Corporation. The awt in 1.0 and 1.1. `http://java.sun.com/products/jdk/awt/`, 2011.

[DA09] Chris Dale and Tom Anderson. *Safety-Critical Systems: Problems, Process and Practice Proceedings of the Seventeenth Safety-Critical Systems Symposium Brighton, UK, 3 - 5 February 2009*. Springer Publishing Company, Incorporated, 2009.

REFERENCES

[Fou11a] Eclipse Foundation. Eclipse - the eclipse foundation open source community website. http://www.eclipse.org/, 2011.

[Fou11b] Eclipse Foundation. Swt: The standard widget toolkit. http://www.eclipse.org/swt/, 2011.

[Fou11c] The Eclipse Foundation. Equinox. http://www.eclipse.org/equinox/, 2011.

[Gee05] David Geer. Eclipse becomes the dominant java ide. *Computer*, 38(7):16–18, 2005.

[GKM82] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17:120–126, 1982.

[HBK04] Marc Hull, Olav Beckmann, and Paul H. J. Kelly. Meprof: modular extensible profiling for eclipse. In *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, eclipse '04, pages 32–36, New York, NY, USA, 2004. ACM.

[Hof11a] Marc R. Hoffmann. Eclemma - jacoco java code coverage library. http://www.eclemma.org/jacoco/index.html, 2011.

[Hof11b] Marc R. Hoffmann. Eclemma - java code coverage for eclipse. http://www.eclemma.org/, 2011.

[Hof11c] Marc R. Hoffmann. The future of code coverage for eclipse. http://www.eclipsecon.org/summiteurope2010/sessions/?page=sessions&id=1745, 2011.

[HS02] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[iCS11] MAP Doctoral Program in Computer Science. Map-i. http://www.map.edu.pt/i/home, 2011.

[JAG09] Tom Janssen, Rui Abreu, and Arjan J. C. van Gemund. Zoltar: A toolset for automatic fault localization. In *ASE '09: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664, Washington, DC, USA, 2009. IEEE Computer Society.

[JHS02a] J.A. Jones, M.J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on*, pages 467 – 477, 2002.

[JHS02b] James A. Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, New York, NY, USA, 2002. ACM.

[JS91] Brian Johnson and Ben Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

REFERENCES

[KTCS05] Bob Kuehne, Tom True, Alan Commike, and Dave Shreiner. Performance opengl: platform independent techniques. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 1, New York, NY, USA, 2005. ACM.

[LBK90] Jean-Claude Laprie, Christian Béounes, and Karama Kanoun. Definition and analysis of hardware- and software-fault-tolerant architectures. *Computer*, 23(7):39–51, 1990.

[LCBR05] J. Lawrence, S. Clarke, M. Burnett, and G. Rothermel. How well do professional developers test with code coverage visualizations? an empirical study. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 53–60, 2005.

[Lou05] P. Louridas. Junit: unit testing and coiling in tandem. *Software, IEEE*, 22(4):12 – 15, 2005.

[oASE11] IEEE/ACM International Conference on Automated Software Engineering. Ase 2009. `https://www.se.auckland.ac.nz/conferences/ase09/`, 2011.

[Pre11] Oxford University Press. Oxford dictionaries online. `http://oxforddictionaries.com/view/entry/m_en_gb0930920#m_en_gb0930920.007`, 2011.

[Ros95] D.S. Rosenblum. A practical approach to programming with assertions. *Software Engineering, IEEE Transactions on*, 21(1):19 –31, 1995.

[Rou11] Vlad Roubtsov. Emma: a free java code coverage tool. `http://emma.sourceforge.net/index.html`, 2011.

[SCGM00] John Stasko, Richard Catrambone, Mark Guzdial, and Kevin McDonald. An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53(5):663–694, 2000.

[SMBM06] Colin Swindells, Karon E. MacLean, Kellogg S. Booth, and Michael Meitner. A case-study of affect measurement tools for physical user interface design. In *GI '06: Proceedings of Graphics Interface 2006*, pages 243–250, Toronto, Ont., Canada, Canada, 2006. Canadian Information Processing Society.

[TAI10] TAICPART. Taicpart 2010. `http://www2010.taicpart.org/`, 2010.

[WF11] Inc. Wikimedia Foundation. Mini-map. `http://en.wikipedia.org/wiki/Mini-map`, 2011.

[WL08] Enhua Wu and Youquan Liu. Emerging technology about gpgpu. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 618–622, 2008.

[XYC05] Zhigen Xu, Yusong Yan, and J.X. Chen. Opengl programming in java. *Computing in Science Engineering*, 7(1):51 – 55, 2005.

[ZL96] Andreas Zeller and Dorothea Lütkehaus. Ddd—a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.

REFERENCES

Appendix A

Installation Guide

GZoltar installation is very easy. First of all, GZoltar is multi-platform. Therefore, you can install it on your Eclipse, regardless your operating system. GZoltar is available to Microsoft Windows, Linux and Apple Mac OS X.

To start the installation process, you have to open Eclipse and go to menu “Help” > “Install new software...”, as presented in the following screenshot:

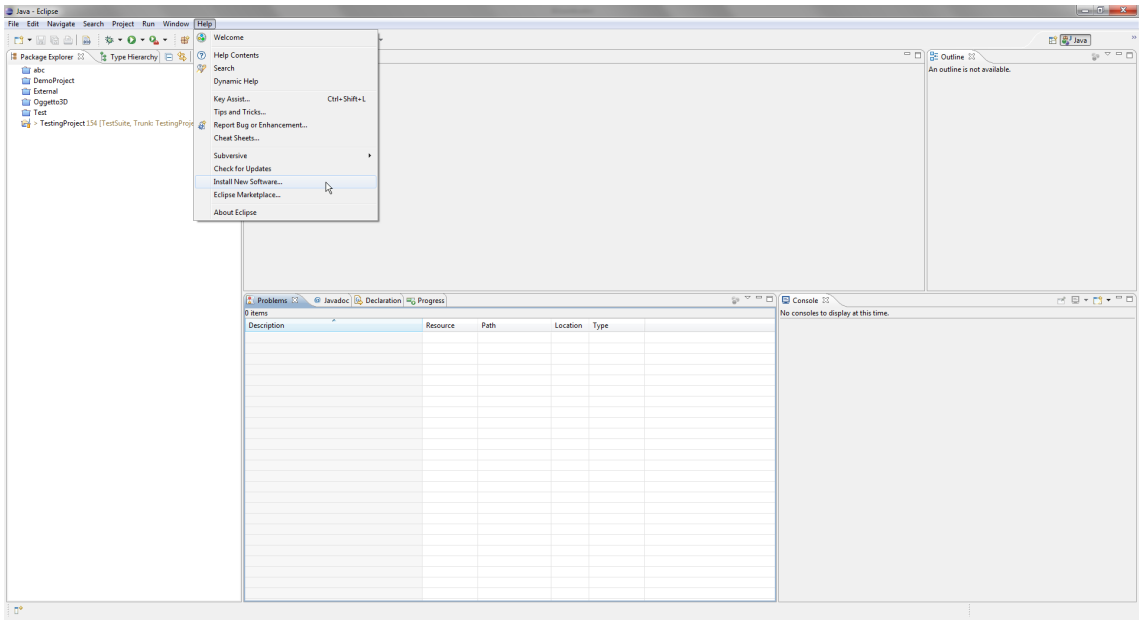

Figure A.1: GZoltar Installation - Step 1

After that, you have to insert GZoltar repository URL (if you do not have it yet) on “Work with” field. GZoltar repository is: http://www.gzoltar.com/

Press “Enter” key, and you will see “GZoltar” category on software list.

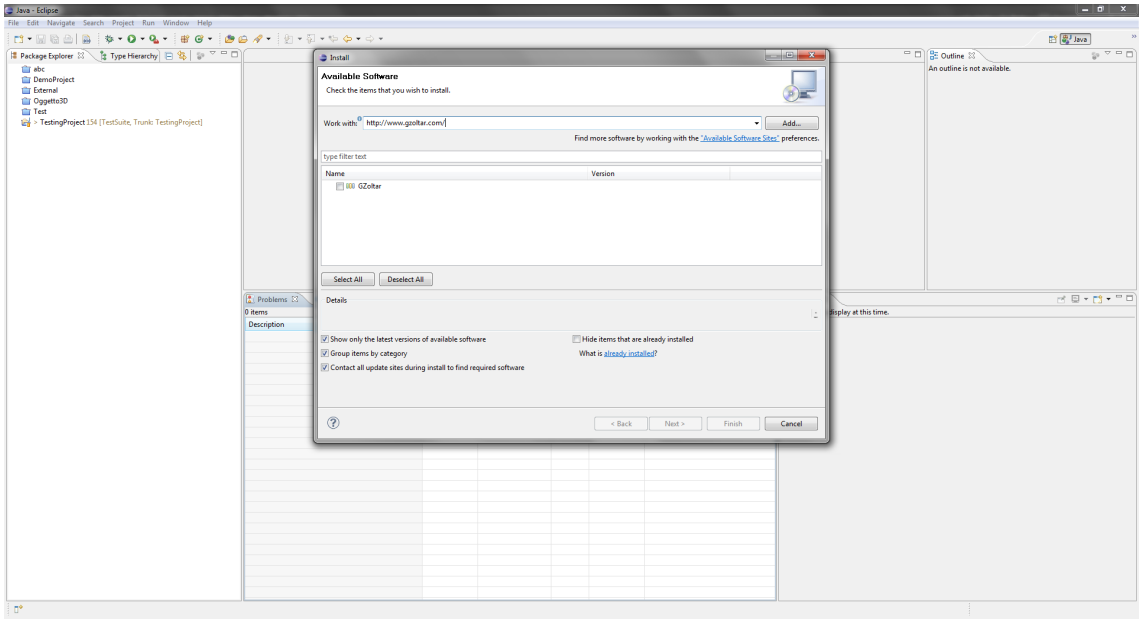

Figure A.2: GZoltar Installation - Step 2

Click on the key proceeding“GZoltar” as presented in the following screenshot:

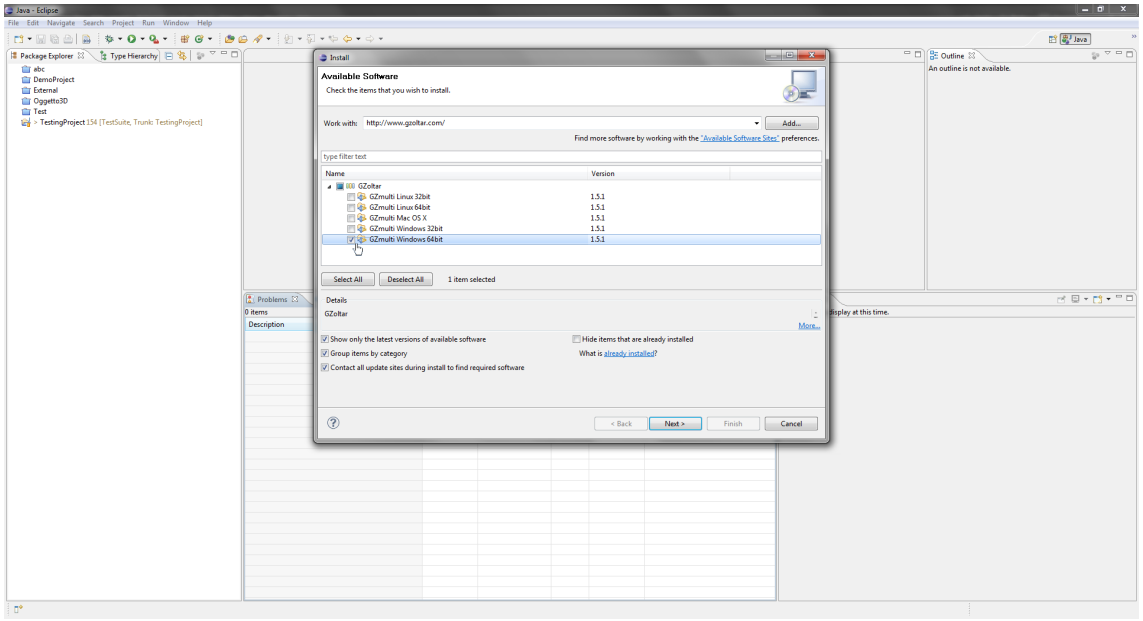

Figure A.3: GZoltar Installation - Step 3

Then, you will see a list of available GZoltar versions. Because GZoltar uses JOGL (OpenGL Bindings) libraries, and they differ from each platform, you have to select your system platform so the correct libraries can be installed along with GZoltar.

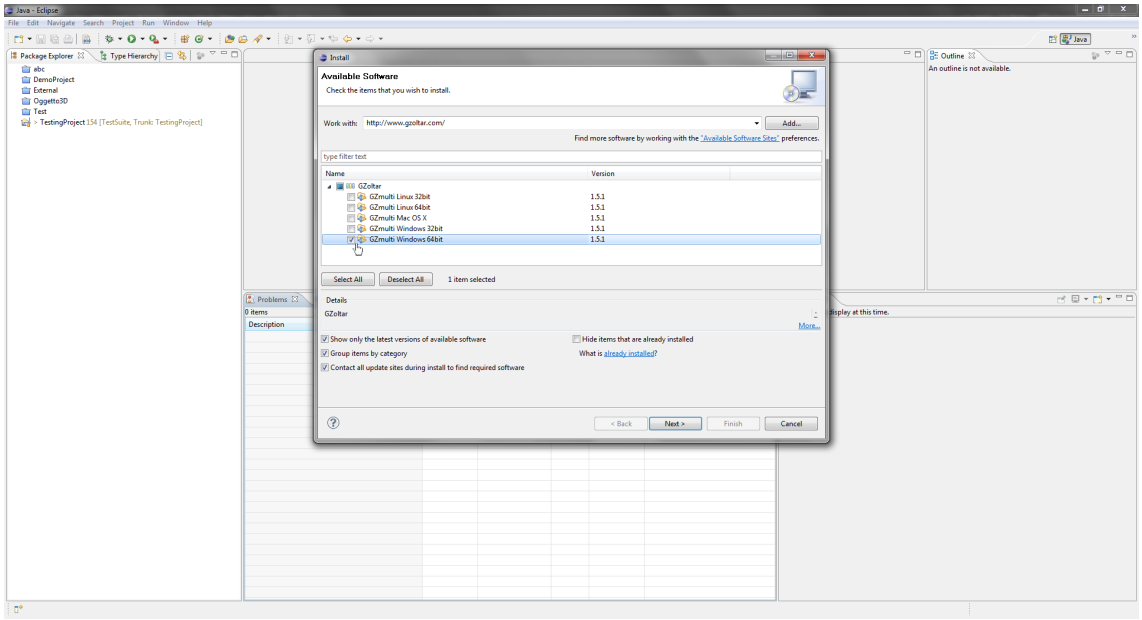

Figure A.4: GZoltar Installation - Step 4

After selected your system, just press “Next >” button to proceed.

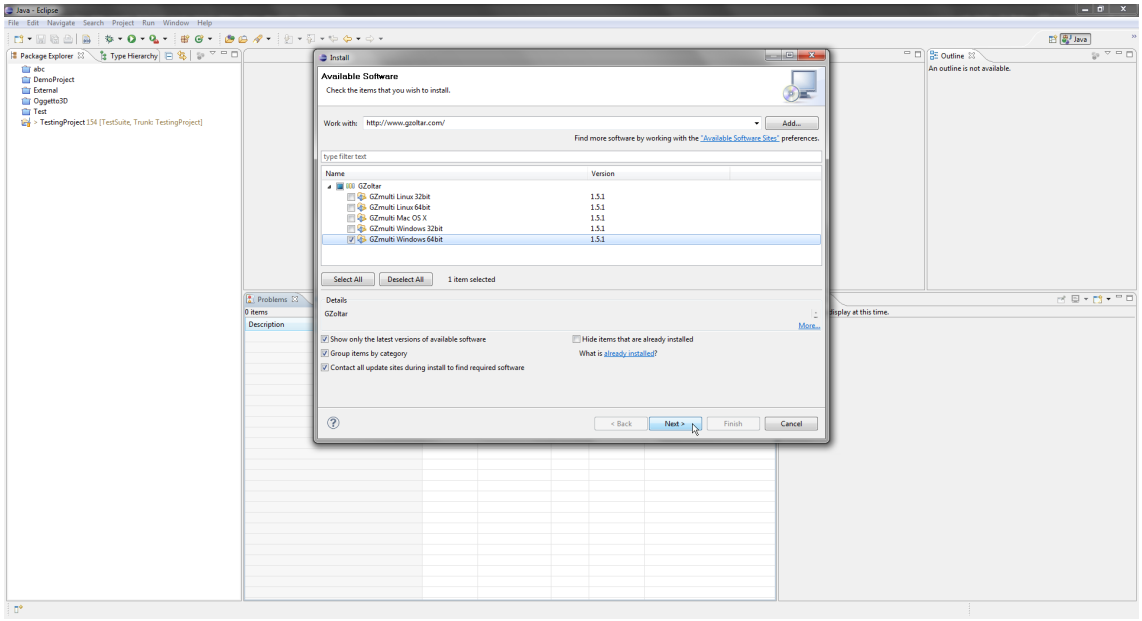

Figure A.5: GZoltar Installation - Step 5

Now you can review your choice. Please confirm again your system and press “Next >” button.

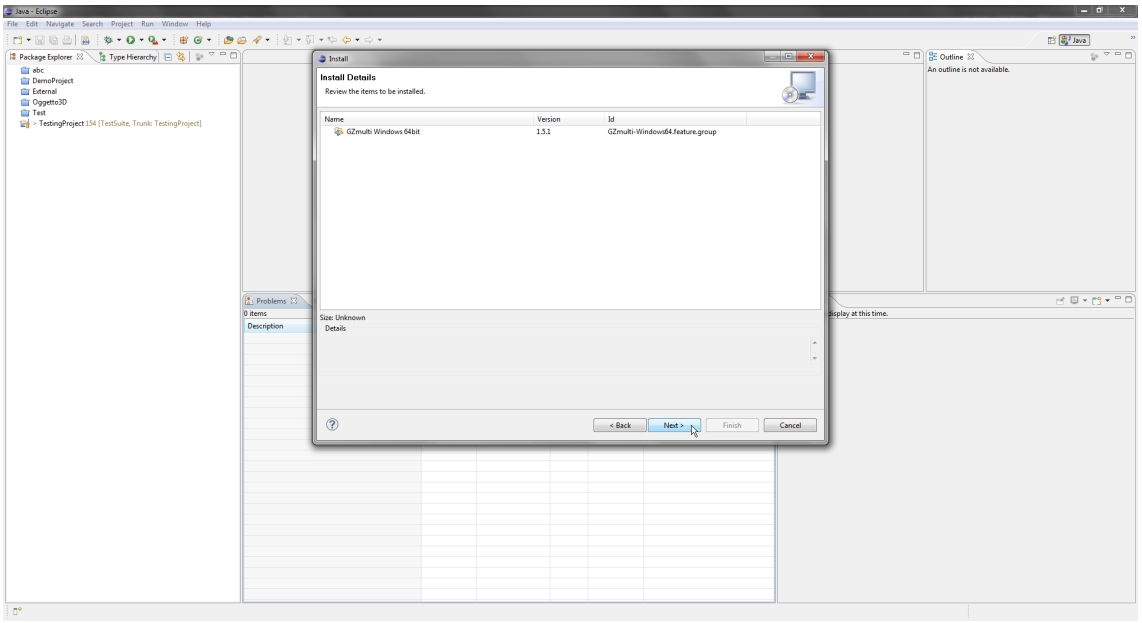

Figure A.6: GZoltar Installation - Step 6

At this point you have to accept GZoltar license to proceed the installation process.

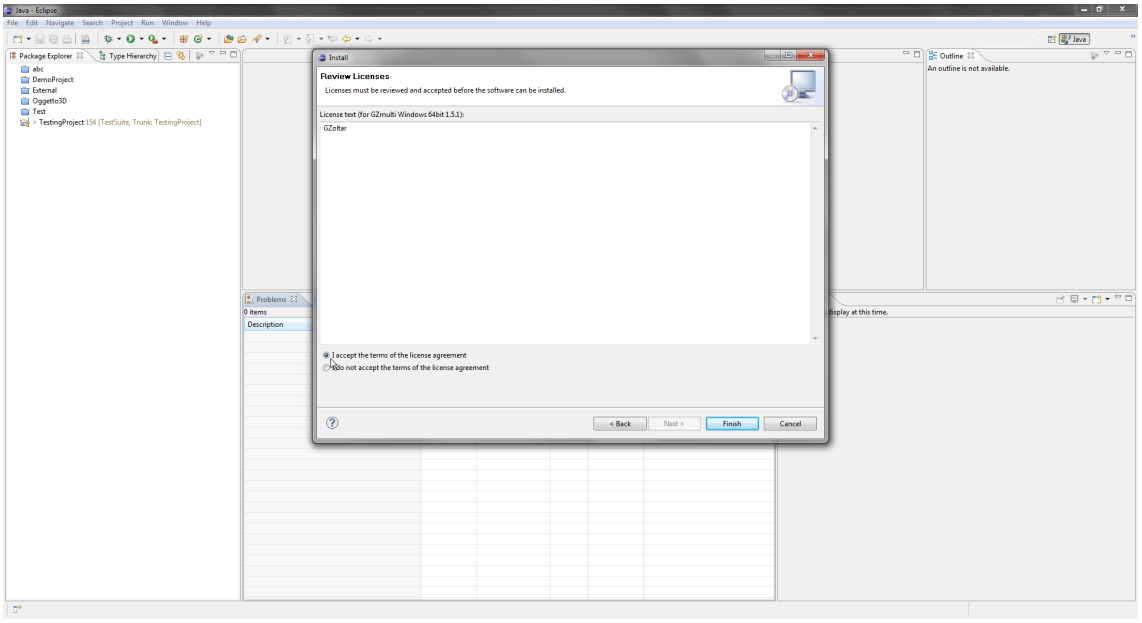

Figure A.7: GZoltar Installation - Step 7

Just press “Finish” button after selected “I accept the terms of the license agreement”. This will start GZoltar installation.

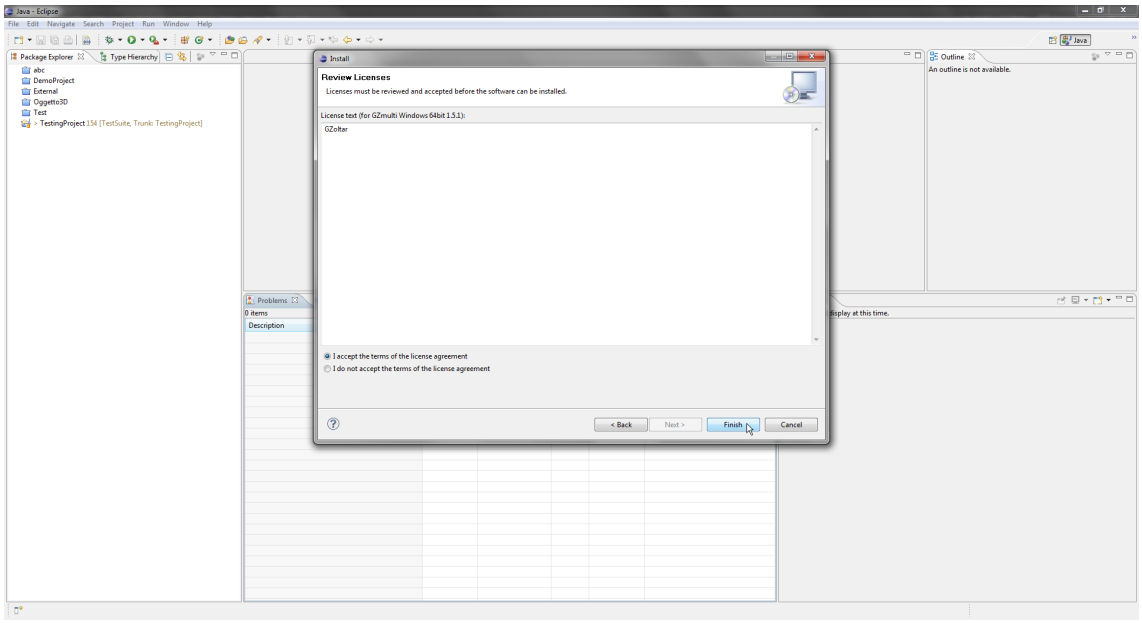

Figure A.8: GZoltar Installation - Step 8

You will get a security warning, because GZoltar is not signed. GZoltar is an ongoing development project and not a released product, that is the reason why GZoltar is not yet signed. Press “OK” button to proceed.

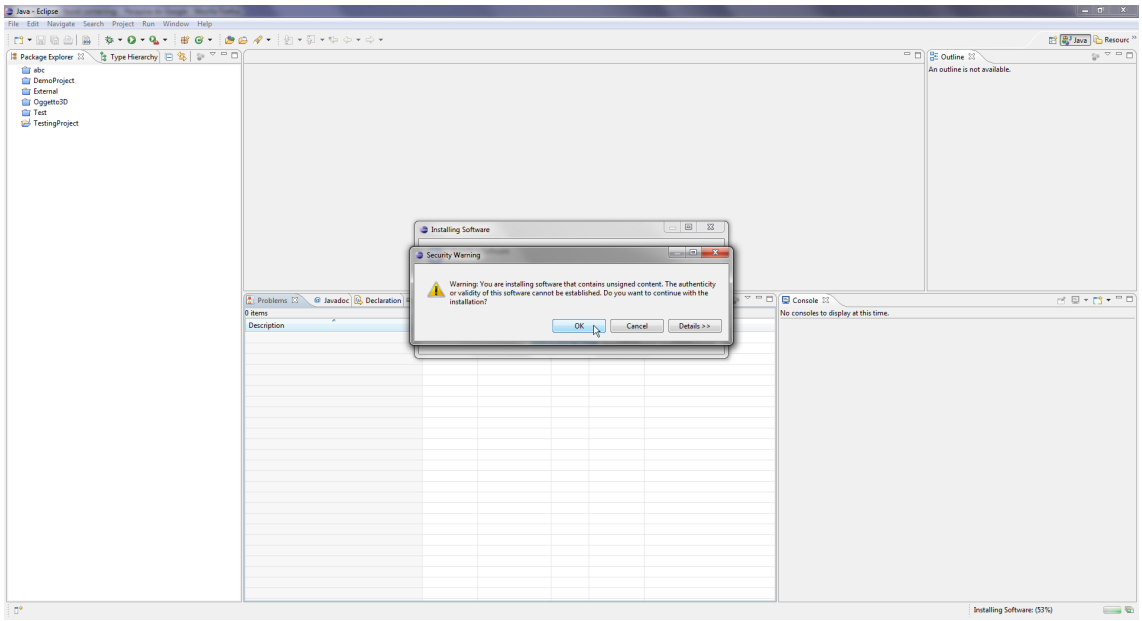

Figure A.9: GZoltar Installation - Step 9

GZoltar will now be downloaded and installed. Just wait until this process is finished.

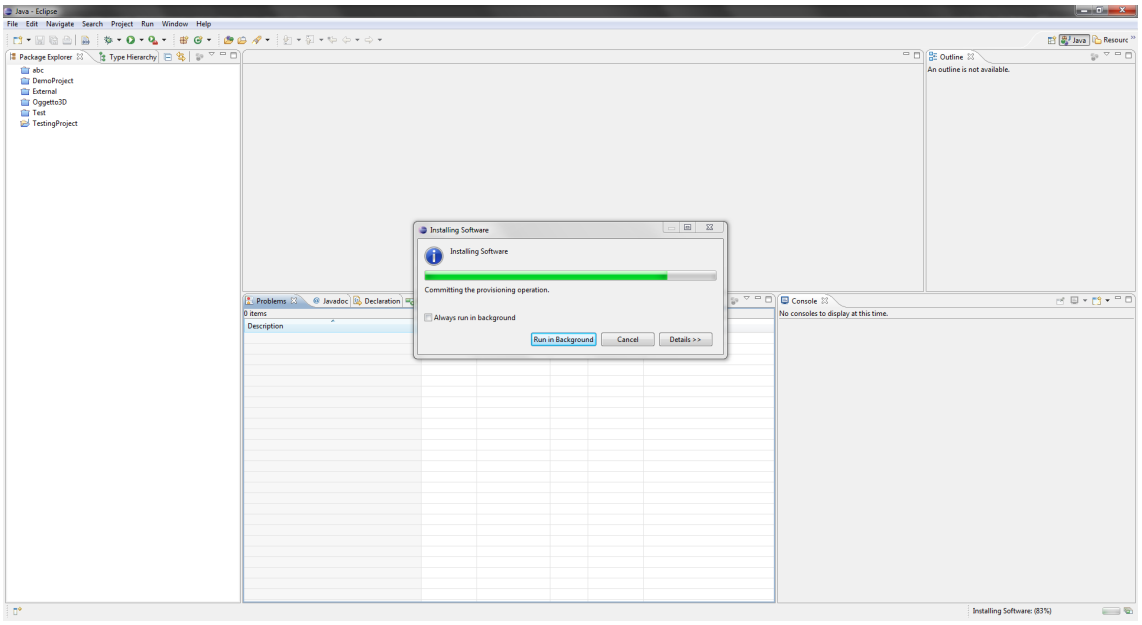

Figure A.10: GZoltar Installation - Step 10

Now it is strongly recommended to restart your Eclipse. Press “Restart Now” button to do it.

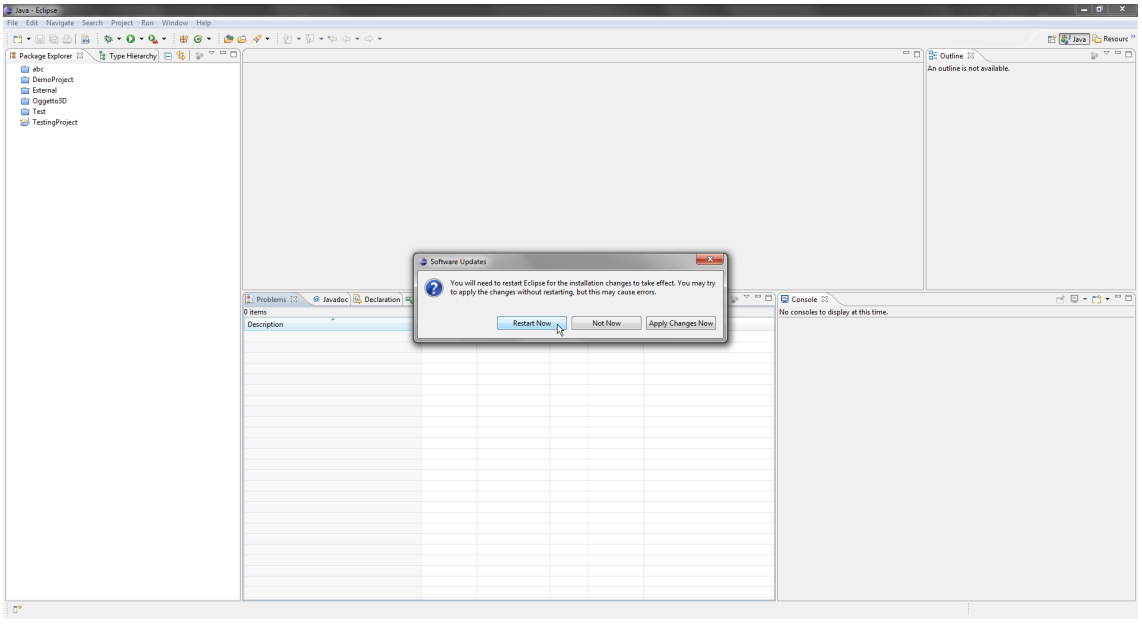

Figure A.11: GZoltar Installation - Step 11

Now GZoltar is installed on your Eclipse. To start it you have to open its view. Go to “Window” menu and choose “Show View”.

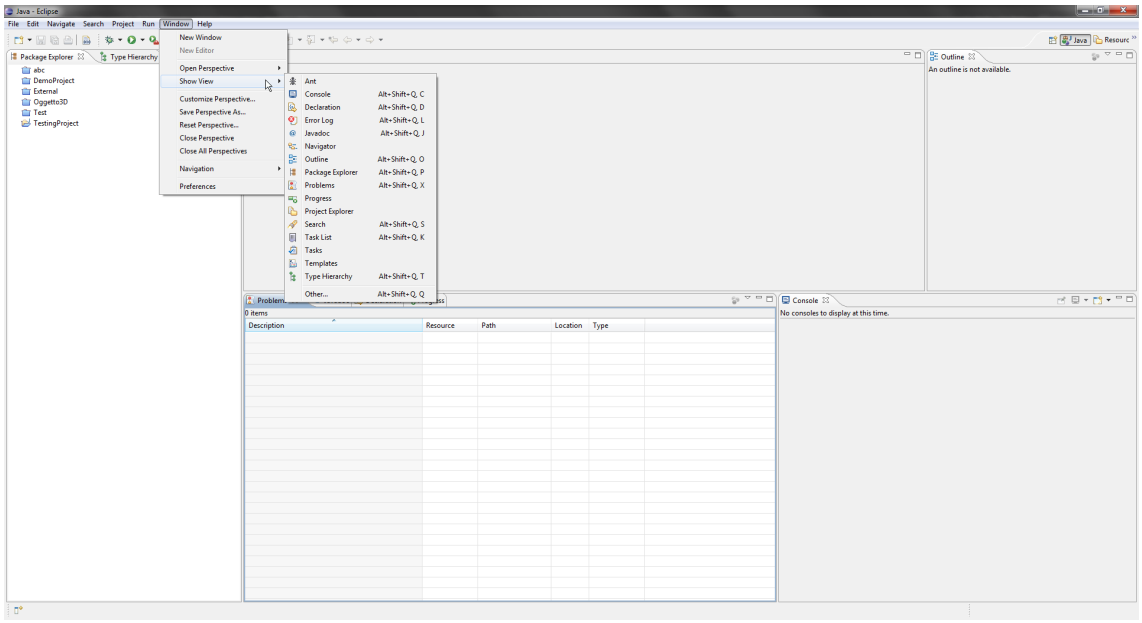

Figure A.12: GZoltar Installation - Step 12

Choose “Other...” on sub-menu, as presented in the following screenshot:

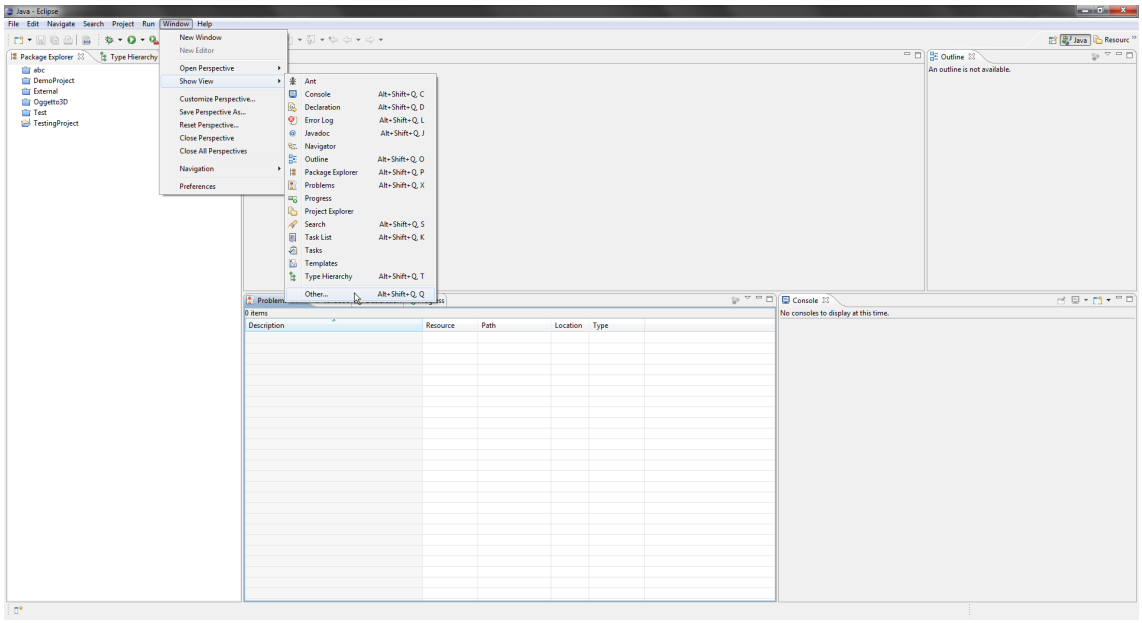

Figure A.13: GZoltar Installation - Step 13

You will now see a list with Views categories. Now you can see a new one called “GZoltar Tools”. Click on the arrow that proceeds the folder icon as shown below:

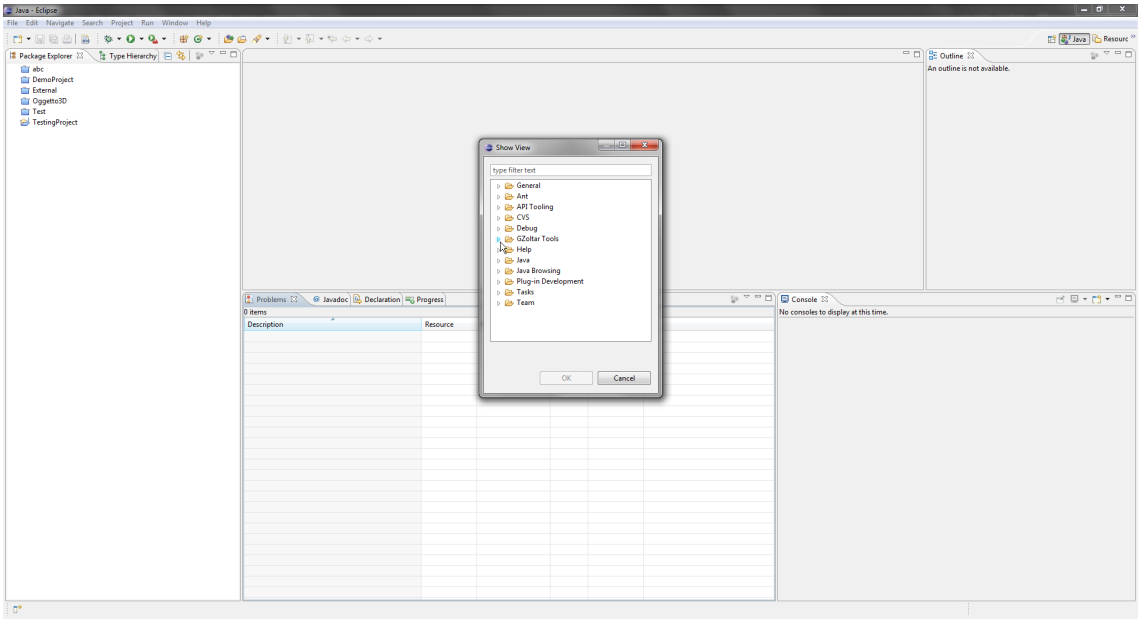

Figure A.14: GZoltar Installation - Step 14

You will now see “GZoltar” view. Click to select it.

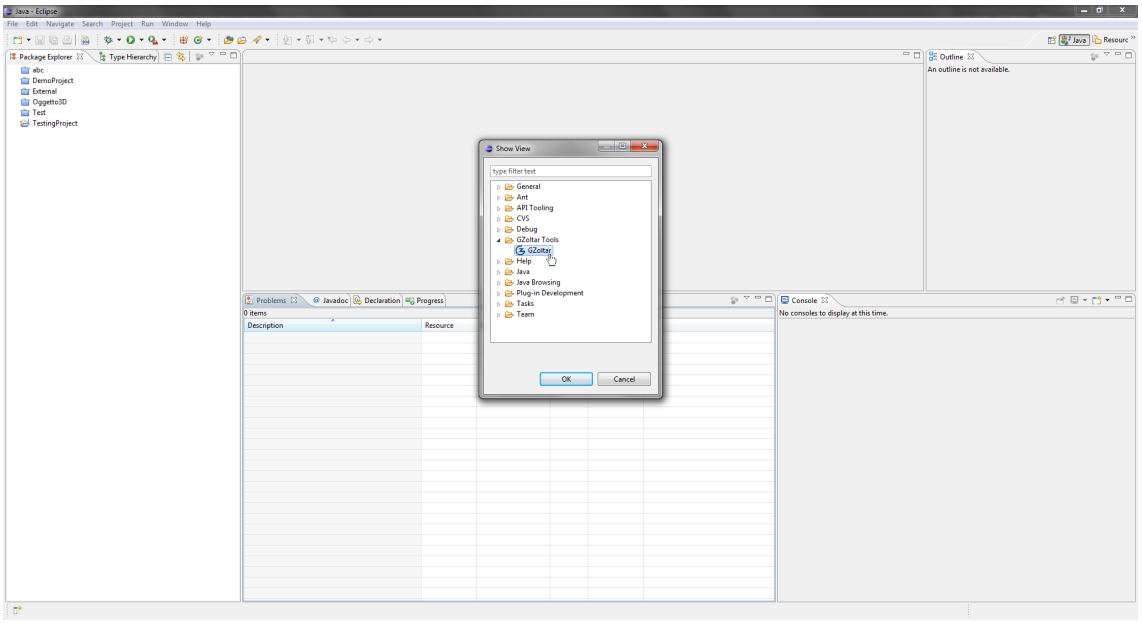

Figure A.15: GZoltar Installation - Step 15

Press “OK” button to accept you want to see GZoltar view. This is the last step of the installation process.

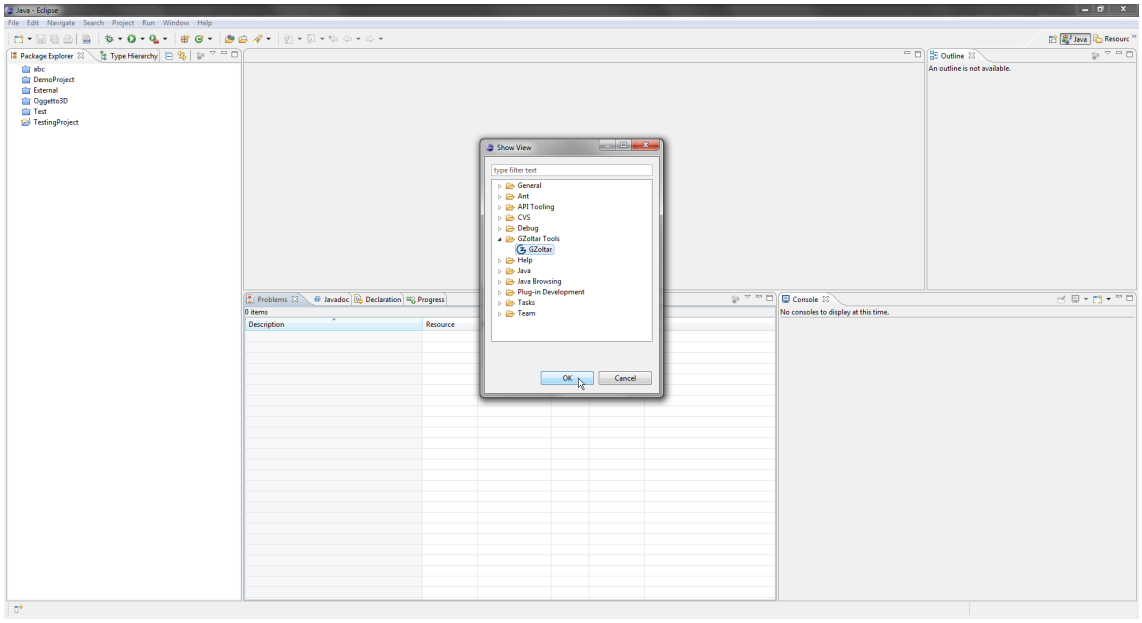

Figure A.16: GZoltar Installation - Step 16

You will then see GZoltar view on its default position. If you don’t have any test class on your opened projects, you will see just a blue rectangle. That is normal. To see GZoltar in action, you have to create a test class.

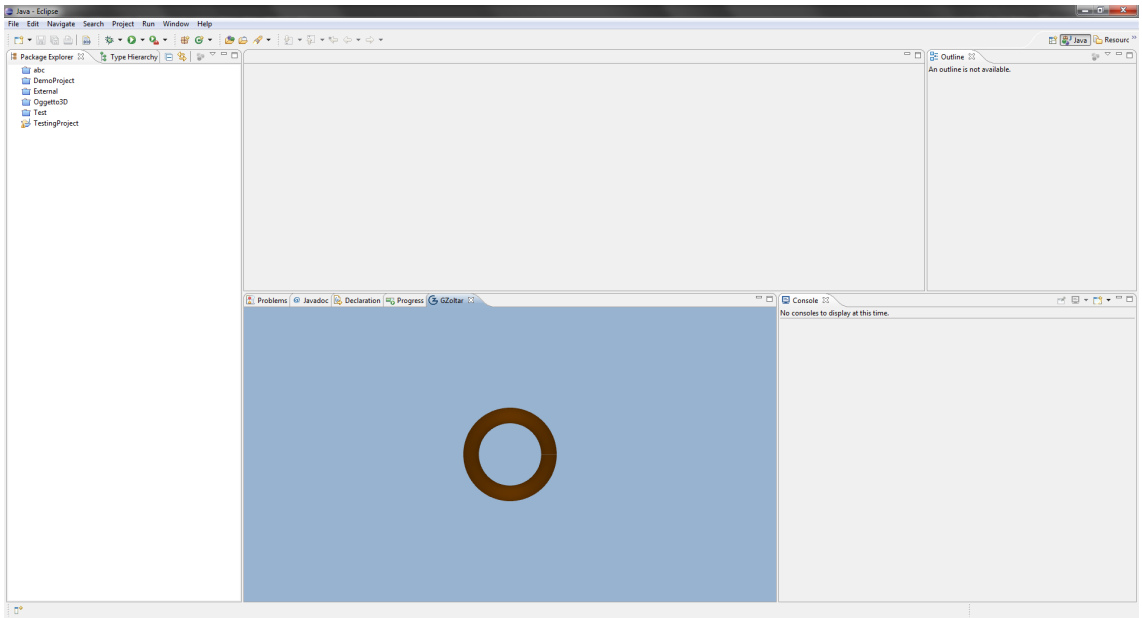

Figure A.17: GZoltar Installation - Step 17

After you have your project with test classes ready, you can select GZoltar view (click on it) and press “F5” key to refresh it. If you want to see all levels expanded, just press “Space” key.

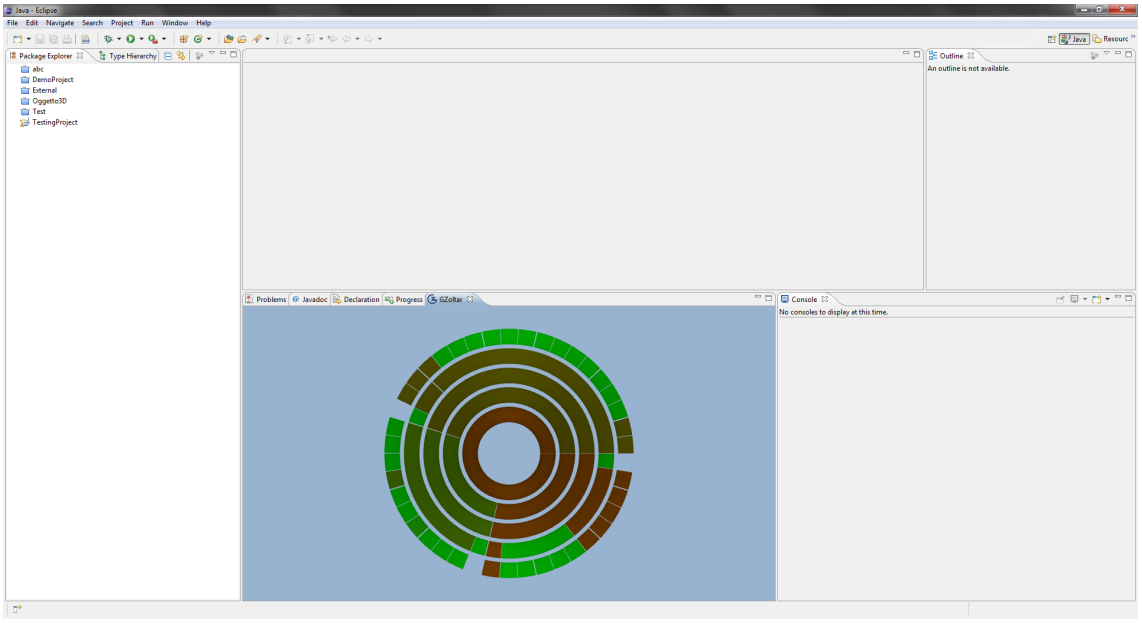

Figure A.18: GZoltar Installation - Step 18

If you prefer Treemap view, you can switch to it at any time by pressing “2” key. You can always switch back to Sunburst if you press “1” key.

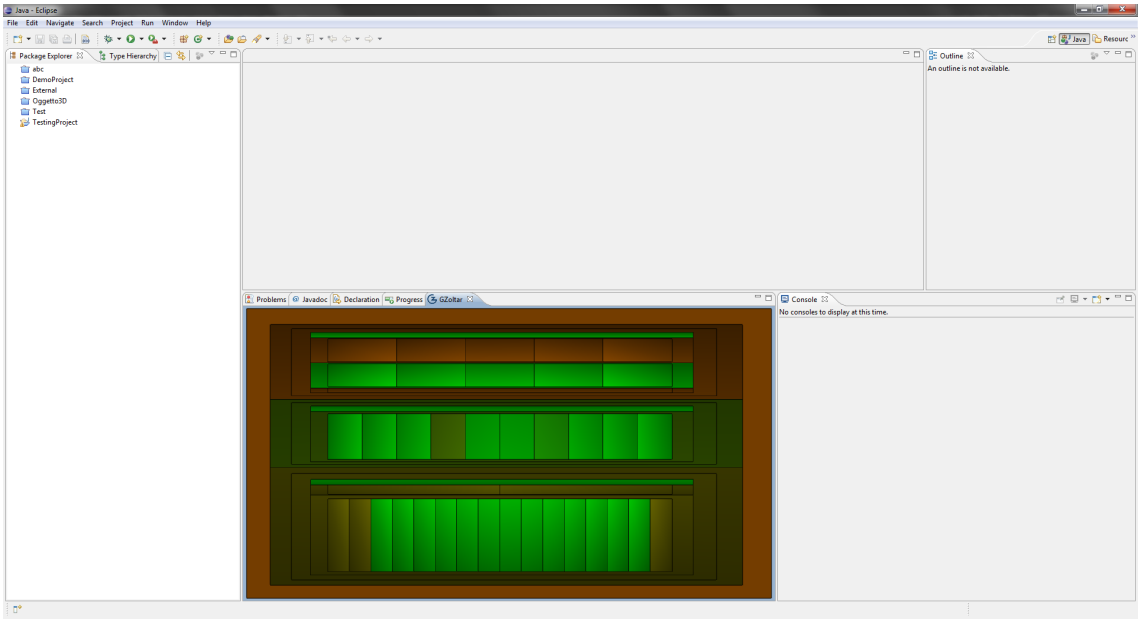

Figure A.19: GZoltar Installation - Step 19

Every time you change your projects code, GZoltar have to update its information. This is not made automatically because of performance reasons. To do so, click on GZoltar view and press “F5” key. You will then see a view with the updated data. If you are working with big projects, it is normal if you have a delay between the time you press “F5” key and have the view ready to navigate.

Figure A.20: GZoltar Installation - Step 20

Appendix B

User Guide

GZoltar is very easy to use. It integrates seamlessly in Eclipse. You just need one or more open projects, and some test classes. Open (if not already) GZoltar view, and then you just need to navigate through the visualizations to quickly find your projects faults.

B.1 Creating Test Classes

The first step to use GZoltar (after installing it) is to create test classes. Test classes are Java classes that implements "Callable<Boolean>". This will allow GZoltar to call that test every time it executes. It should have a "call()" method that returns a boolean value, indicating if the test passed (true) or not (false).

Test classes must have a name finished in "Test", for example: "DemoTest.java".

Here is an example of a test class:

Class 2 Demo GZoltar test class in Java.

```
import java.util.concurrent.Callable;

public class DemoTest implements Callable<Boolean>
{
   @Override
   public Boolean call() throws Exception
   {
      // <Insert your test code here>
      boolean testResult = (...);
      // <Return test result (boolean) at the end>
      return testResult;
   }
}
```

That is all you need to do. Create a class for each test you want to do. You can define here the granularity of the tests you want to make. You can have a test (or more) for each method, or you can have tests that test an entire block of your project. GZoltar works better (produces more accurate results) with a low granularity test stack.

B.2 Eclipse Problem List

GZoltar automatically creates Eclipse warnings and informations, so you will be able to see them on the same place as usual: The “Problems” view. You can sort blocks by failure probability, if you sort this list by name.

You can also jump directly to the referred line, by double-clicking on it.

GZoltar produces standard Eclipse warning notices, so you can work with them just the way you are used to with any other warning message.

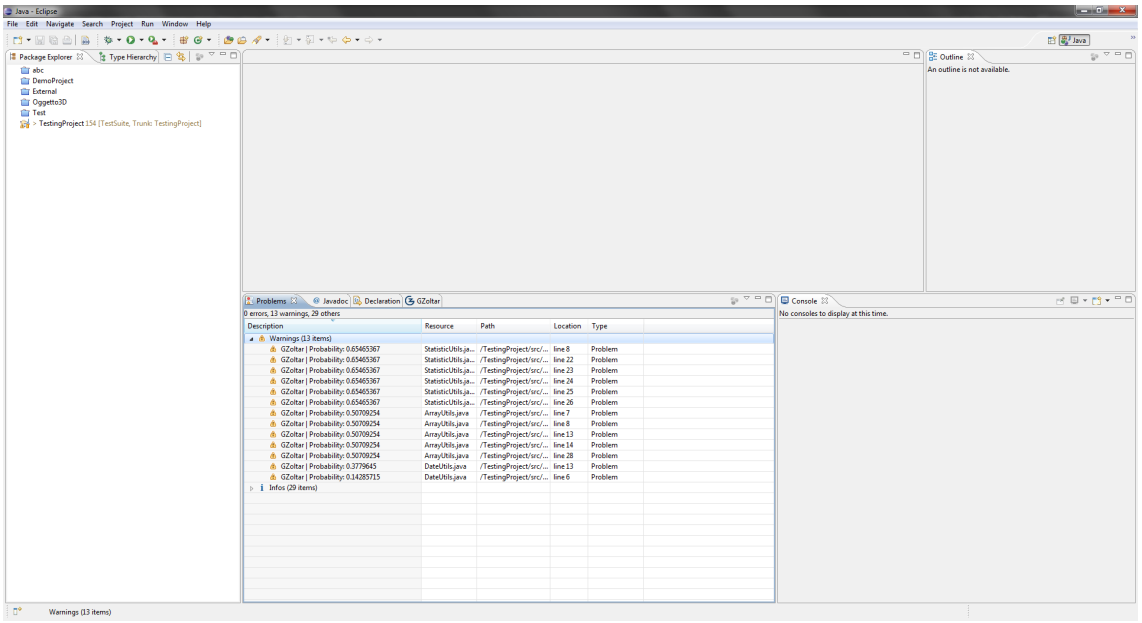

Figure B.1: Eclipse Problems List

B.3 Eclipse Code Editor

You can jump directly to the desired line of code when you are exploring the project structure in your favorite visualization. You just have to click on a leaf node to open the editor directly on the wanted file and line.

With this feature, you can quickly and easily switch between your fault localization task and your fault correction one, so you will be able to all the debugging process at the same place, in the environment that you are used to work in: you IDE.

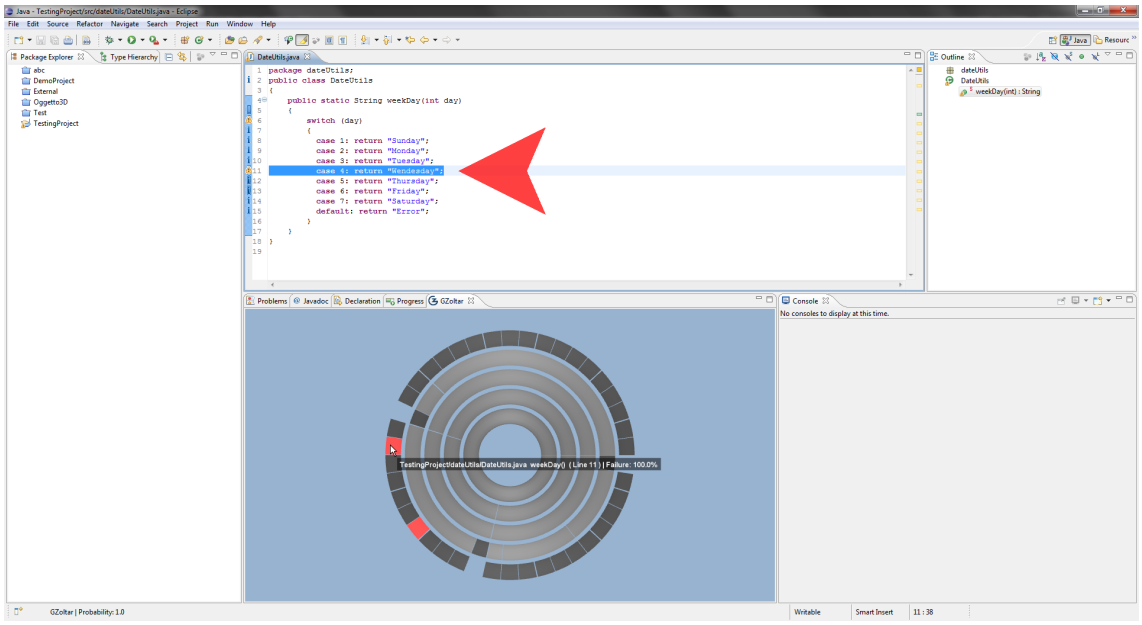

Figure B.2: Open Code Editor through GZoltar view

At the code editor, you will have traditional Eclipse warning tooltips that will help you to get information about GZoltar failure probability for each executed line of code.

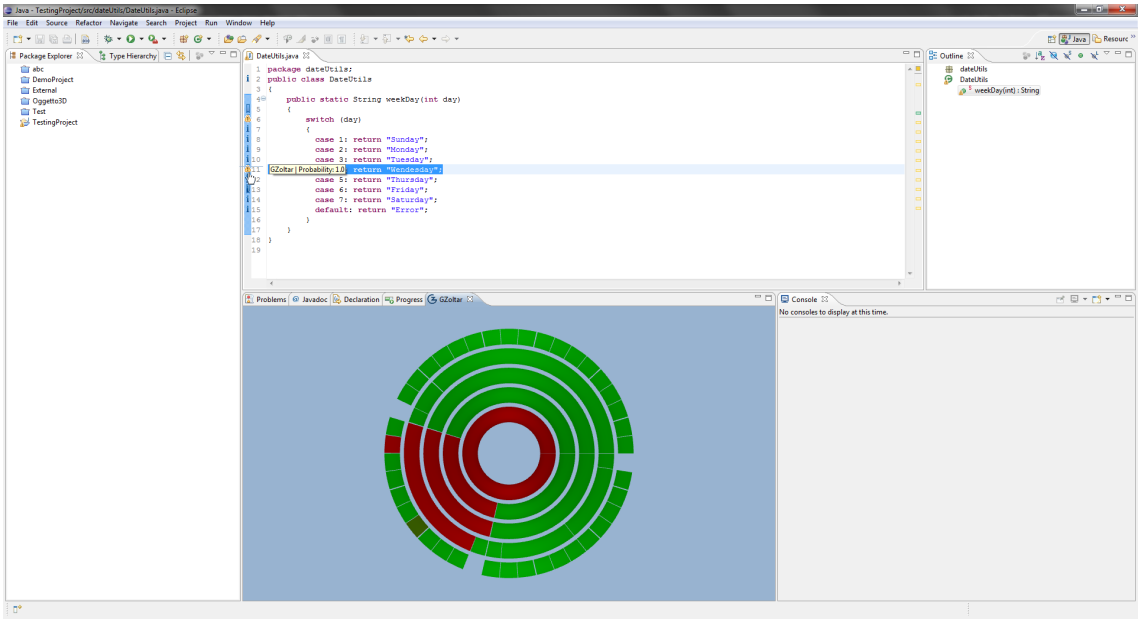

Figure B.3: Eclipse Editor Tooltip

You can have the same tooltip information on Eclipse editor scrollbar, as you are probably used to (if you are an Eclipse user).

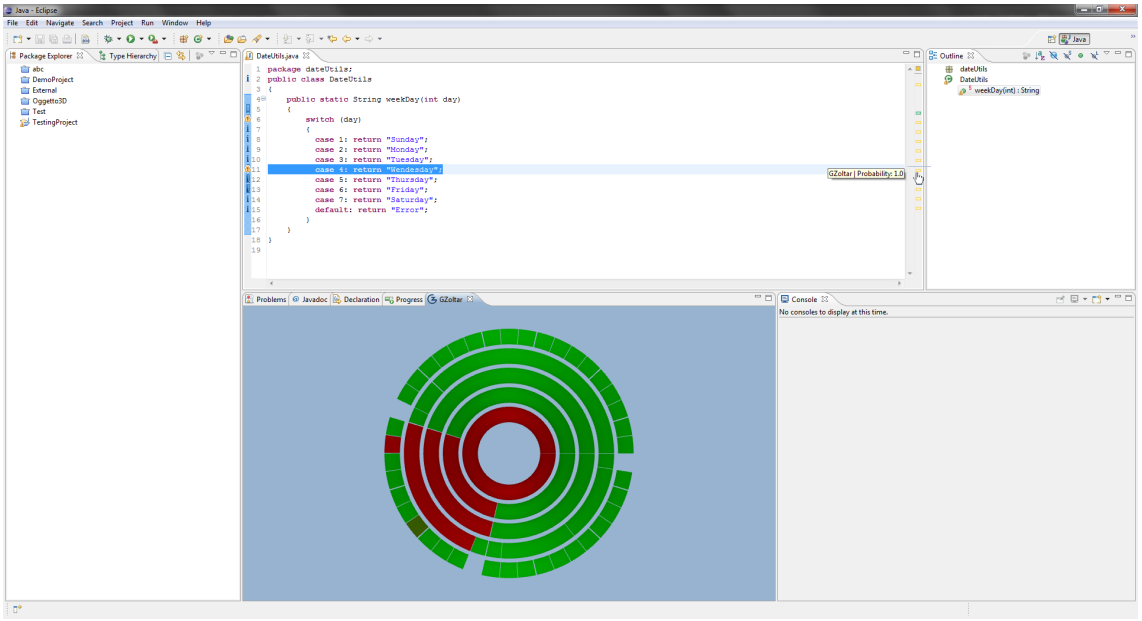

Figure B.4: Eclipse Editor Scrollbar Tooltip

B.4 Visualizations

The two visualizations that this first GZoltar version have are just for example purposes. GZoltar goal is to be a platform to have many other visualizations, to help the developer in is fault localization task.

All visualizations share the same principals. We will present the two example visualizations, so you can better understand GZoltar capabilities.

We present here the two visualizations side-by-side, so you can even compare them and see which you prefer.

Please note that you can choose the visualization you want by pressing the key with their correspondent number. Sunburst is the default visualization, so if you press "1" key, you will switch to Sunburst. Pressing "2" key you will switch to Treemap. Your tree structure is maintained when you switch your view. Even if you did a root change, it will be maintained on view switch.

B.4.1 Navigation

You can navigate through any GZoltar visualization. By default, all levels are collapsed. You can click on a node to open its sub-nodes.

This principle is common to all visualizations. Every time you click on a node, regardless its shape or location, you will expand all its sub-nodes if that given node is an inner one, or will open code editor if it is a leaf one. In this case it is an inner node, so if you click on it, you will expand its contents.

When you place mouse cursor over a node, you can see it name and its failure probability.

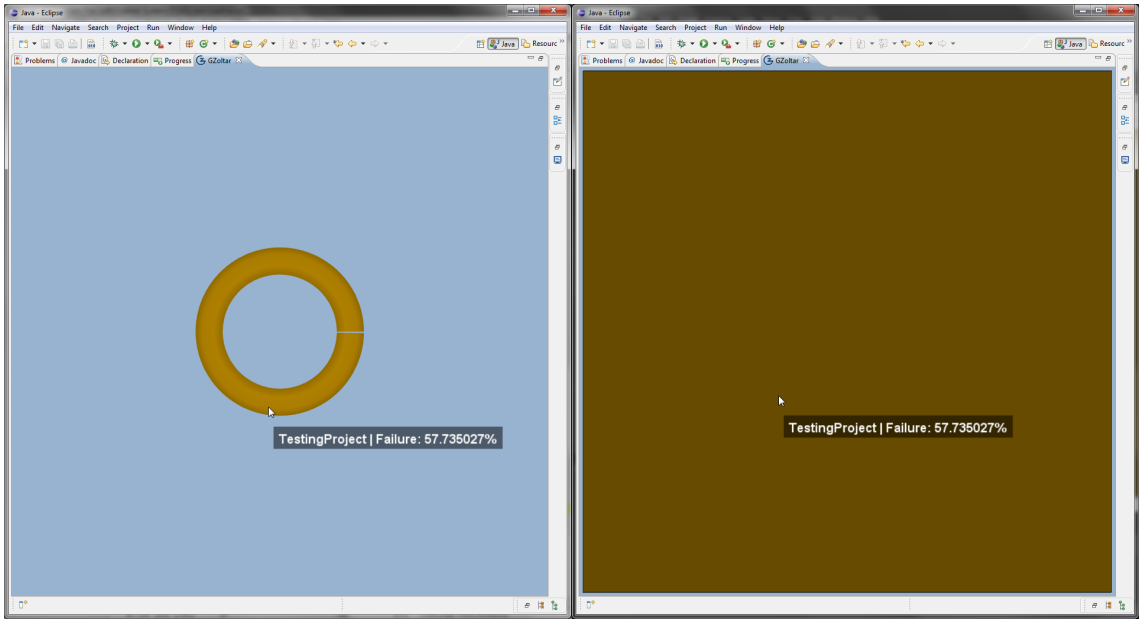

Figure B.5: navigation - Step 1

This is also common to all visualizations. Every time you place your mouse cursor over a node, no matter what is its shape or location, you will see a tooltip with the full name of that node (like a system file path). If the node is a method, or even a line of a class or method, you will also see that information in that tooltip.

You can also see GZoltar failure probability. In inner nodes, this value represents the maximum value of their sub-nodes. In leaf nodes, this value represents that exact component failure probability, calculated according to the SFL algorithm that is being used by GZoltar.

In this example, you have three sub-nodes. If you click on the inner node, you will collapse it again.

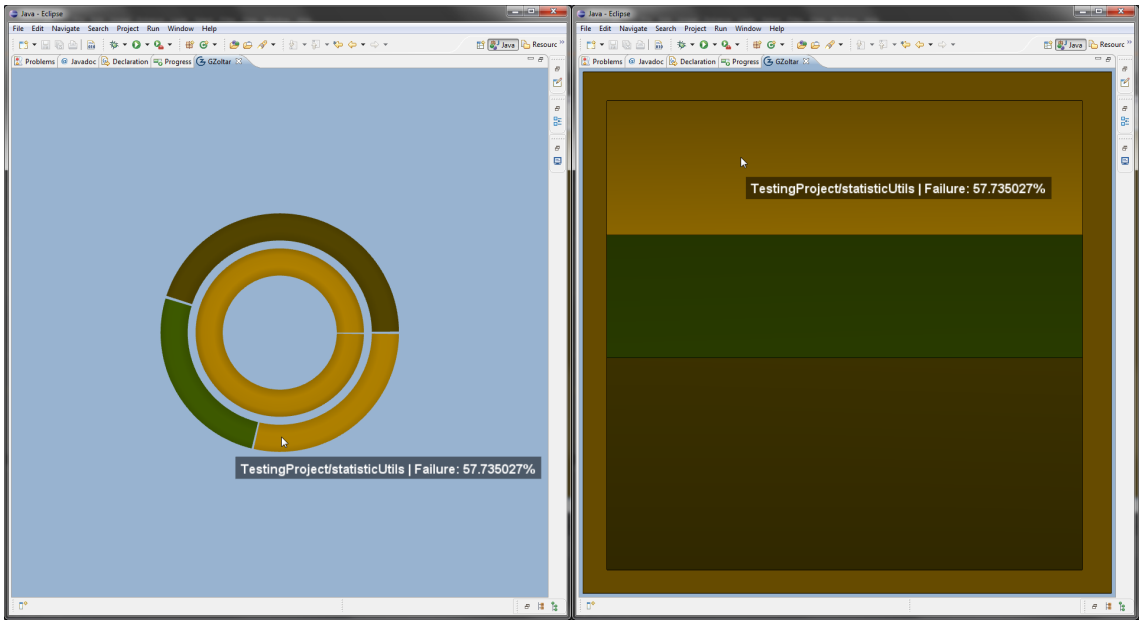

Figure B.6: Navigation - Step 2

You can click on any node to expand its sub-nodes (or collapse them if already expanded).

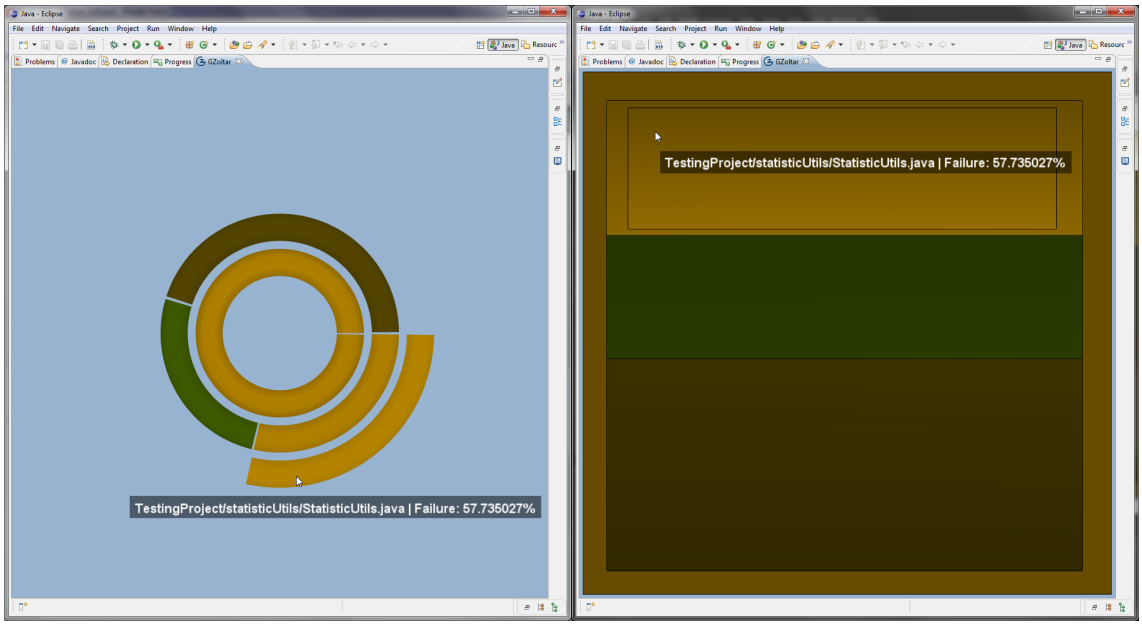

Figure B.7: Navigation - Step 3

That way you can navigate in the hierarchy of your project. At any level you have the full path of the selected node, to help you in its localization. The values presented on upper levels (and their colors) represents the maximum of its sub-levels.

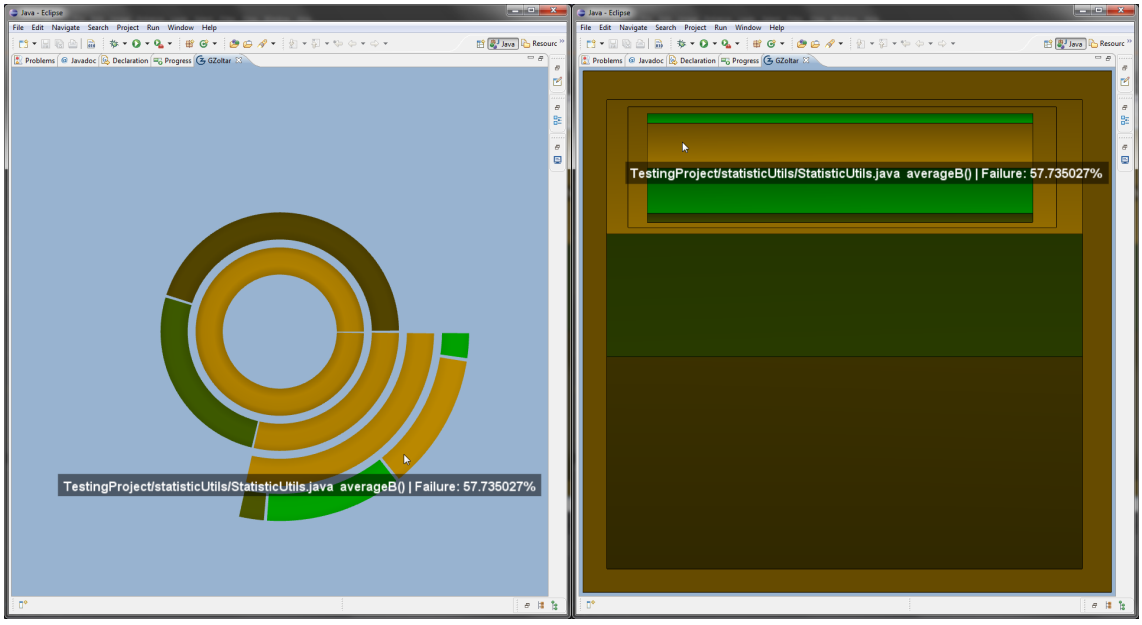

Figure B.8: Navigation - Step 4

At the last level, on the tree leafs, you can click on the node to open the code editor on the correspondent file and line, to access immediately to that piece of code. When you have your mouse cursor over a node that is placed at the last level, the colors you see are changed from an individual failure probability to a global one, based on the selected node. If the node you selected was never executed during tests, you will see a complete gray sunburst. If it was executed, you will see its color representing its failure probability. Every leaf nodes will have a color that represents its relation to that node. If a node was executed every time the selected node was executed (as in the following figure), it will have exactly the same color as the selected one. Otherwise, it will have a color that vary between the selected node color (100% of simultaneous execution) and dark gray (0% of simultaneous execution).

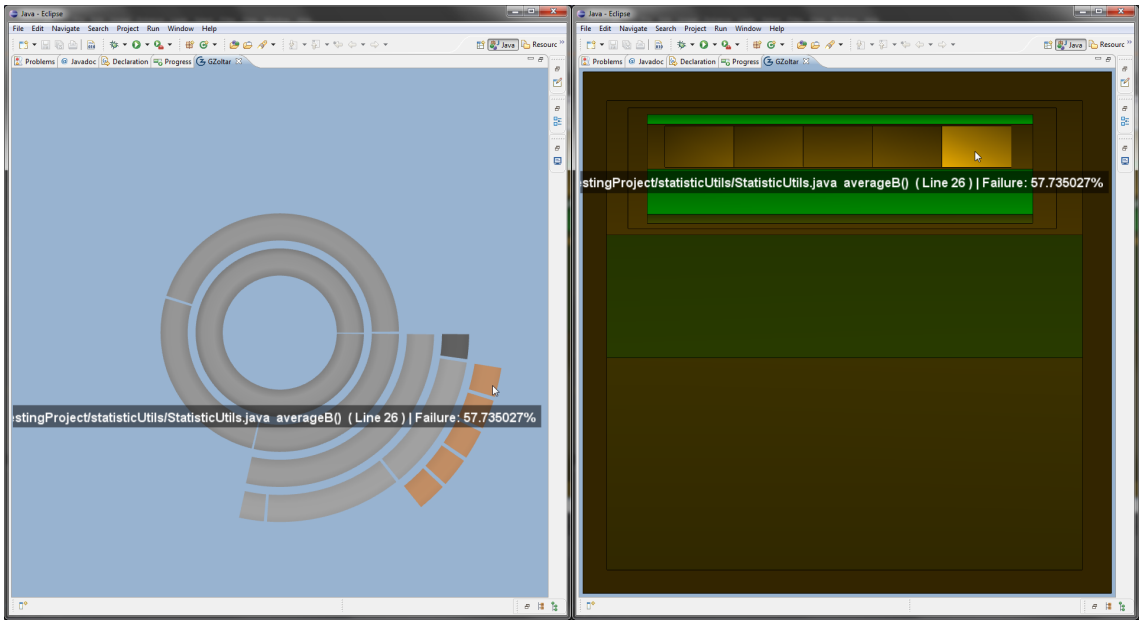

Figure B.9: Navigation - Step 5

At any time you can press “space” key to expand all levels and see the whole system.

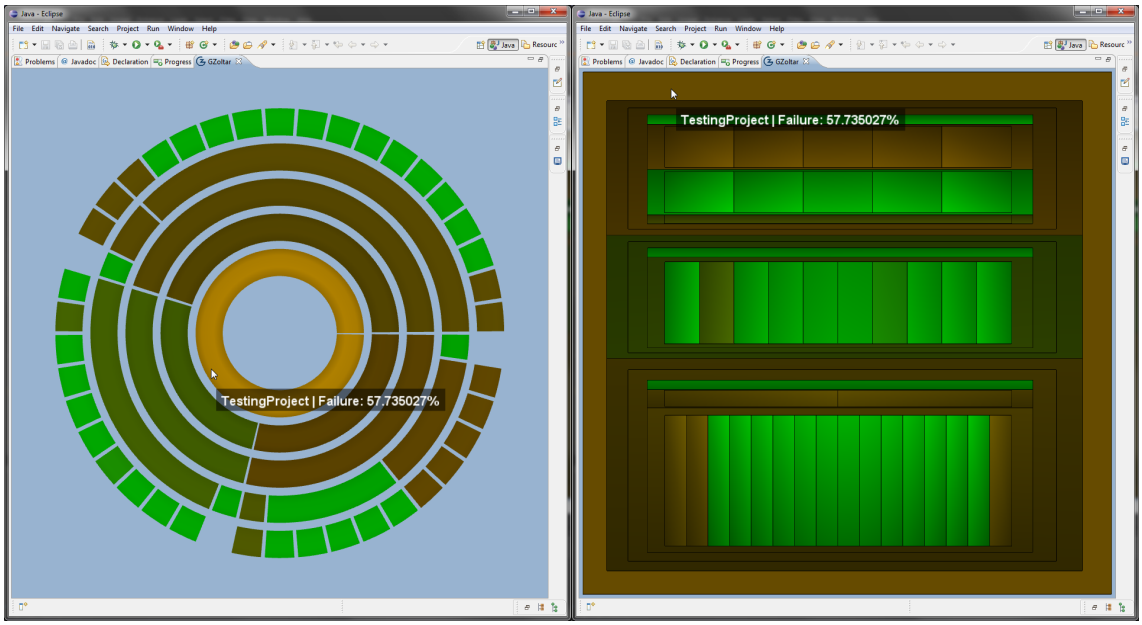

Figure B.10: Navigation - Step 6

B.4.2 Zoom and Panning

There are times when you can have a pretty confusing visualization, with very tiny nodes. To aid your visualization, you can zoom in and pan to the area you want to see in detail.

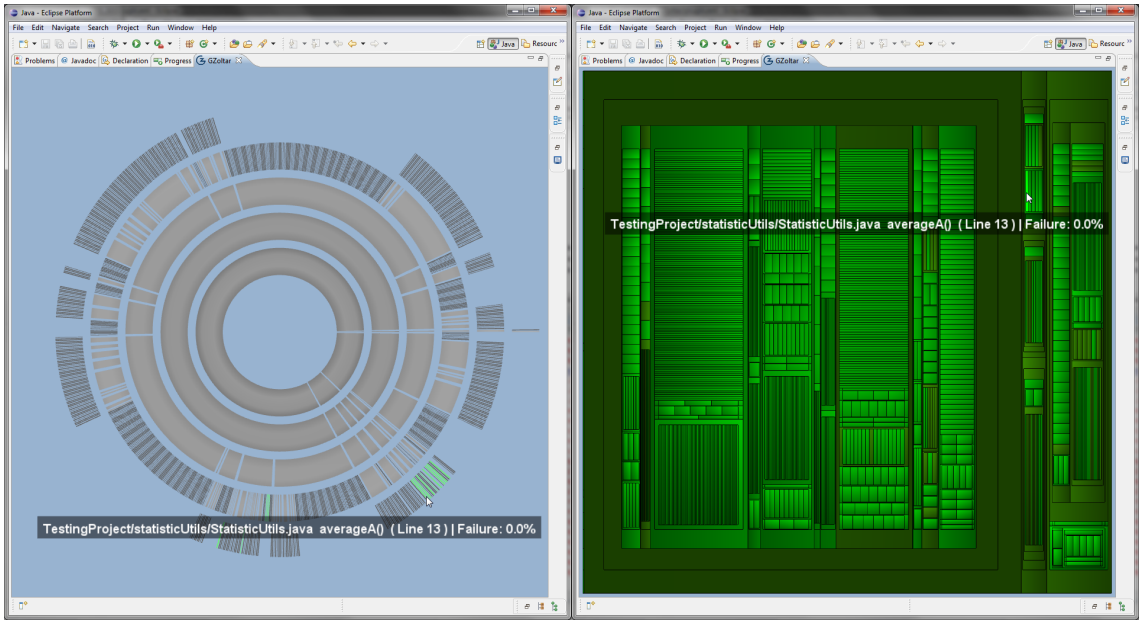

Figure B.11: Navigation - Big Projects

To zoom in you can double click with your mouse on any area of GZoltar view, but don't release the mouse button on the second click. You will zoom in until you release the mouse button or you reach to the maximum zoom level. You can pan the visualization by clicking on it (at any point) without releasing the mouse button. You can move your mouse to reposition the view. After that, you just have to release the mouse button to stop the panning process.

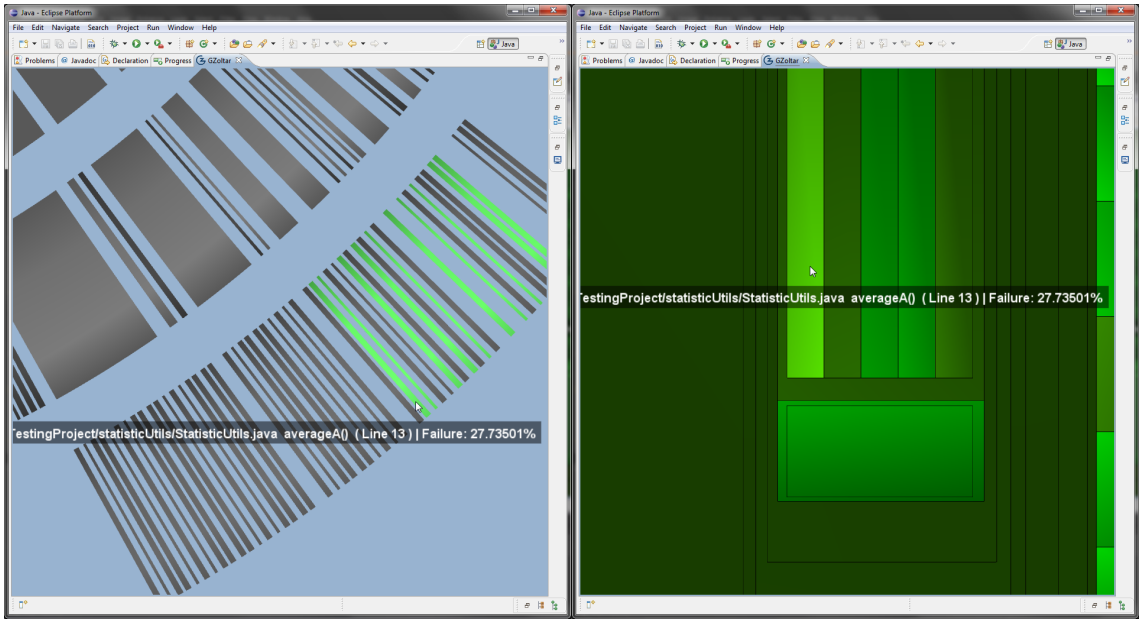

Figure B.12: Navigation - Zoom and Panning

B.4.3 Root Change

If you want, you can change your visualization root to any inner node of your tree. To do that, all you have to do is to click with the right mouse button on the node you want to focus. You will then see a new visualization that will discard all nodes that are unrelated to the selected one. You will only see the selected node, its siblings, and their sub-nodes.

In big projects, this is a very useful feature, because you can analyze with maximum detail any area of your system.

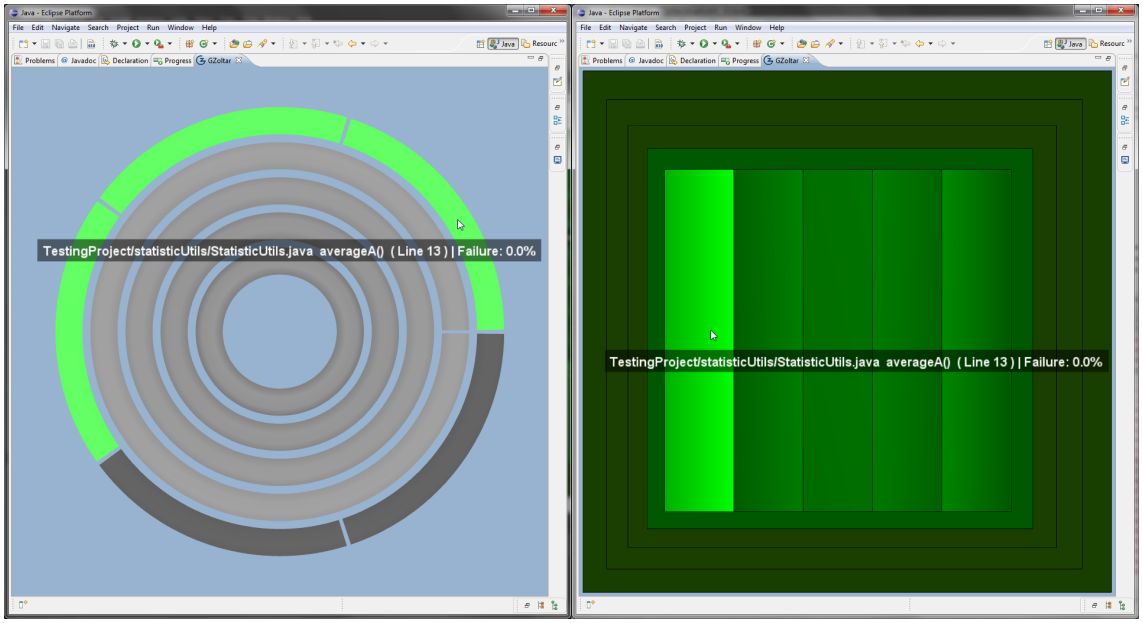

Figure B.13: Navigation - Zoom and Panning

B.4.4 Faults Correction

GZoltar is very well integrated with Eclipse IDE. Therefore, you can quickly alternate between your coding process and your project visualization. Every time you want to see an updated version of your project (after changing its code) you just have to click on GZoltar view and press “F5” key. We will present an example of a debugging process. In this example we have a project with tree different faults, and we will correct them one at time.

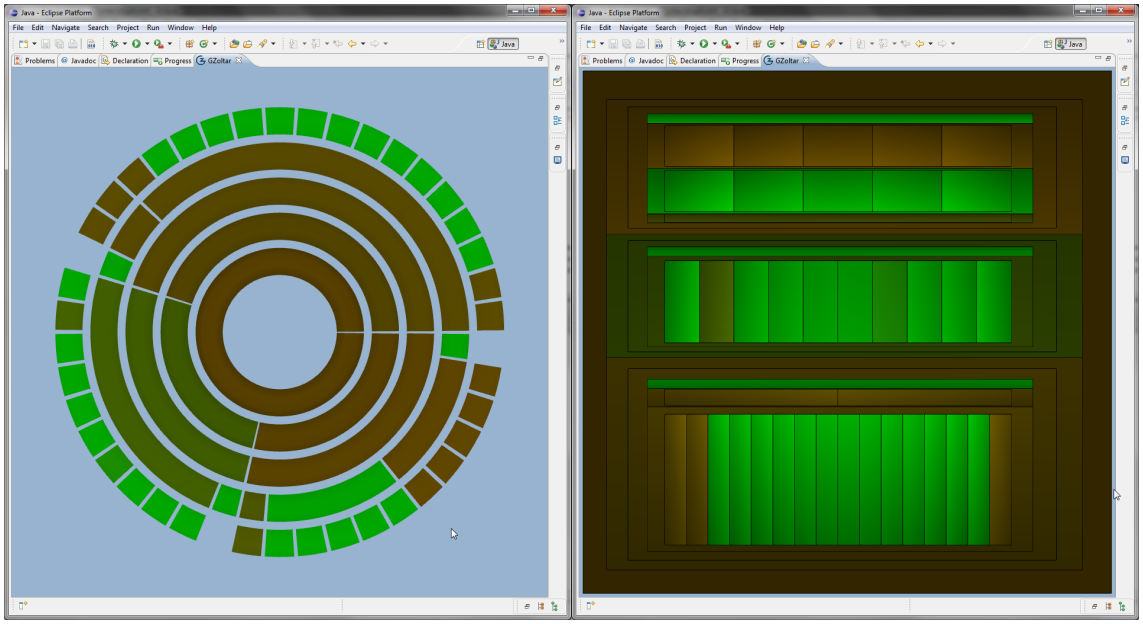

Figure B.14: Fault correction - Start

By clicking on the leafs, we can jump immediately to its code.

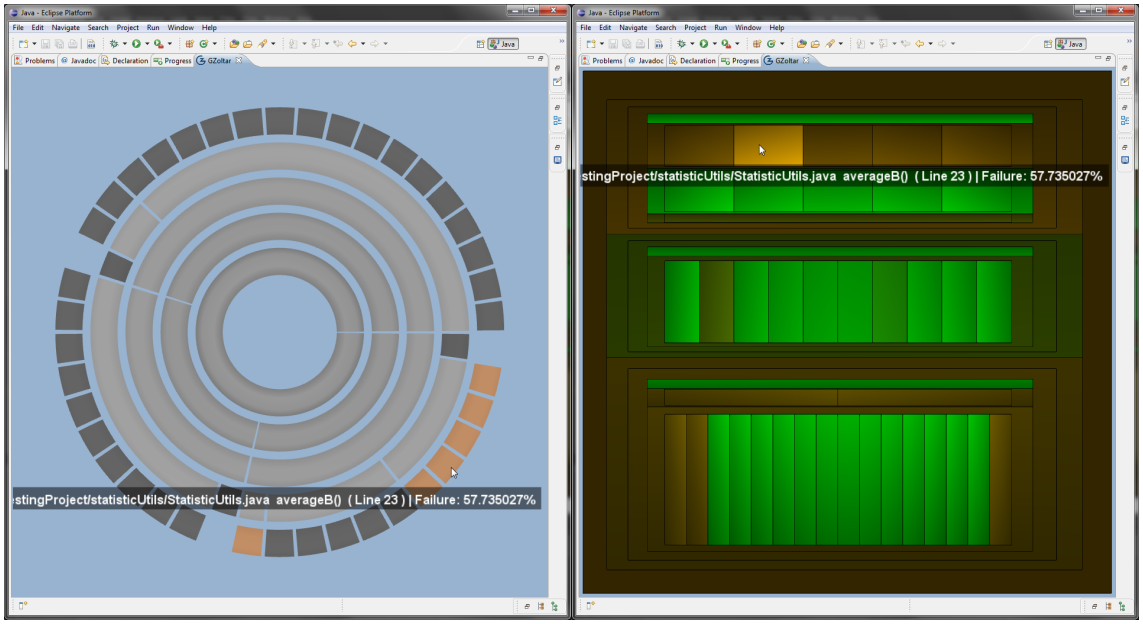

Figure B.15: Fault correction - Step 1

We can then correct that fault, and refresh GZoltar view (“F5”). We will then see the updated view. As you can see, GZoltar accuracy gets higher as you correct the faults of your system.

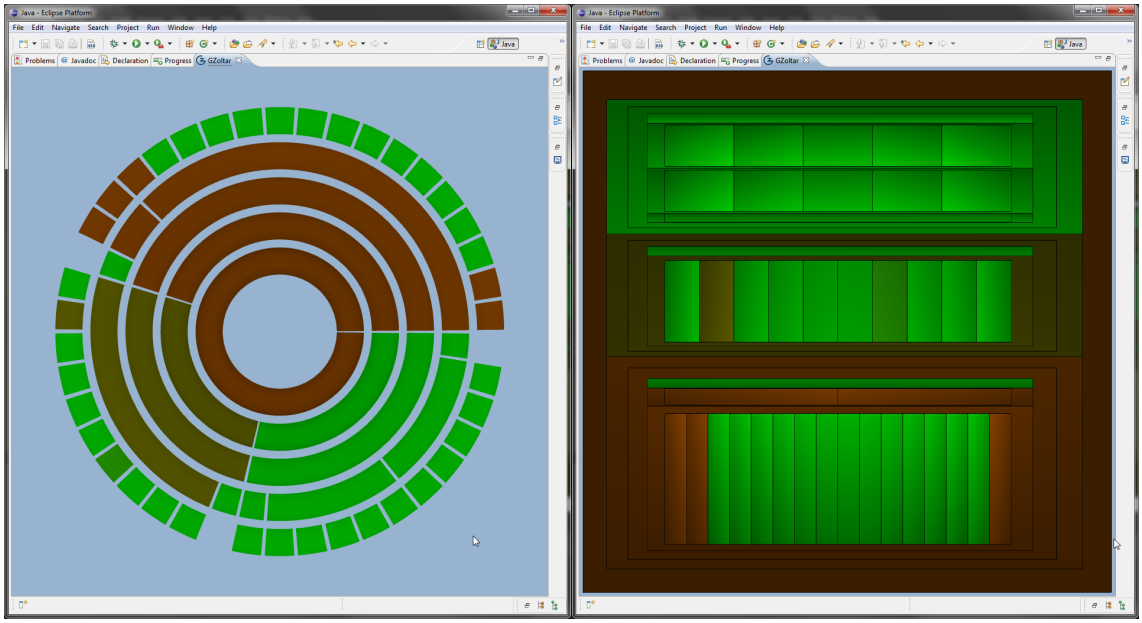

Figure B.16: Fault correction - Step 2

Once again, click on the desired node to open the editor directly on that file and line.

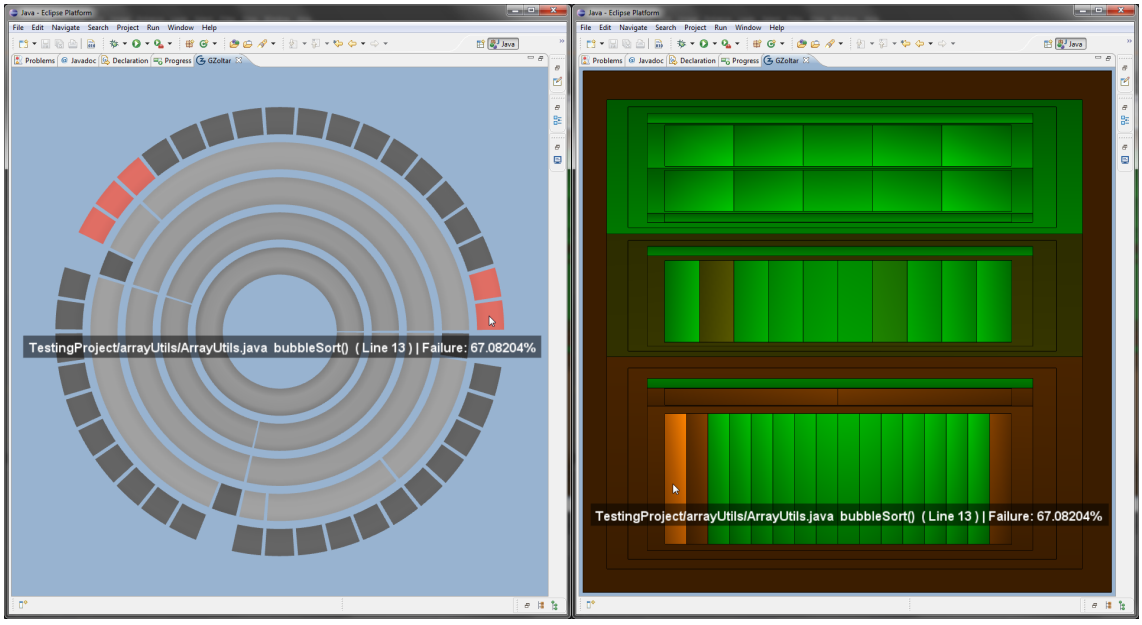

Figure B.17: Fault correction - Step 3

We can correct another fault, and update GZoltar view again. Every time you correct a fault, GZoltar accuracy gets higher.

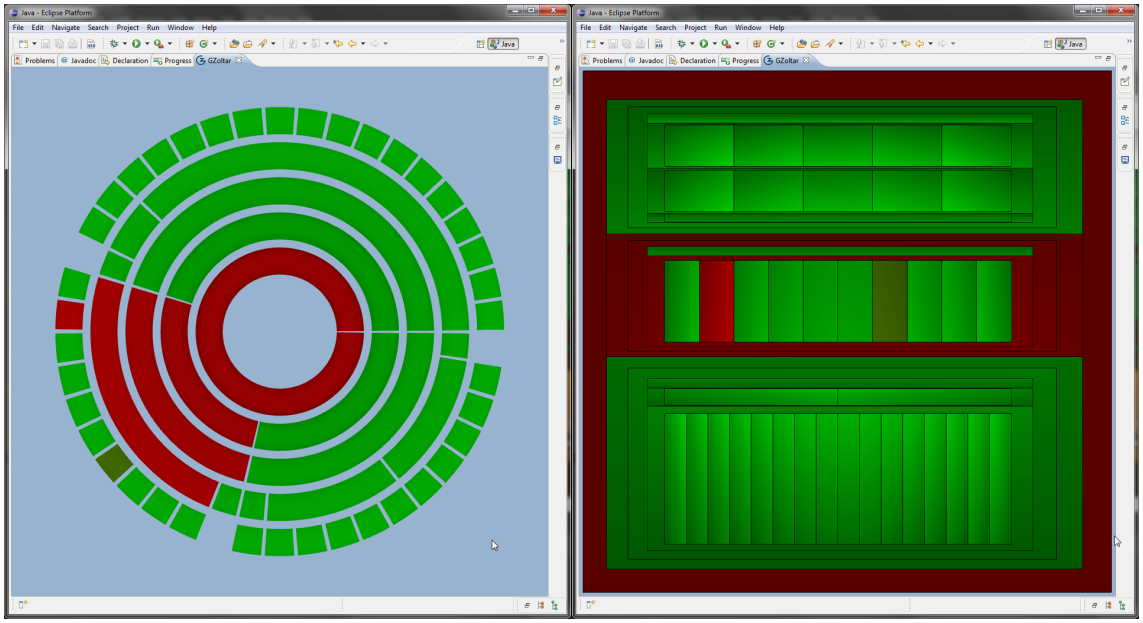

Figure B.18: Fault correction - Step 4

Click on the red node to open the editor. This is the last fault.

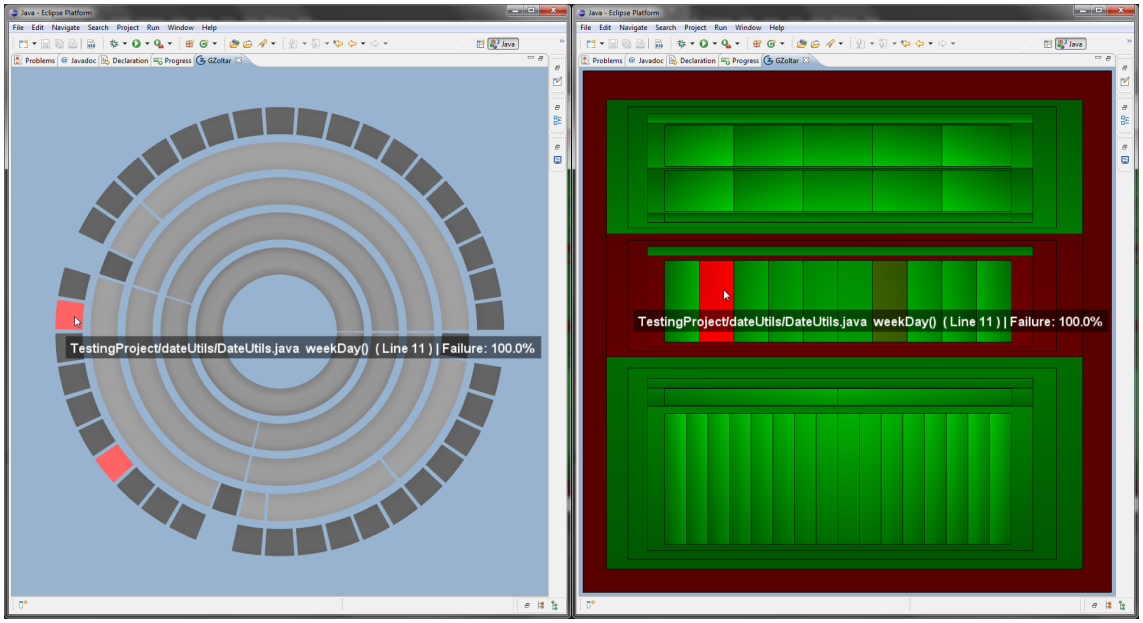

Figure B.19: Fault correction - Step 5

At the end, all your tests will pass, and no fault will be detected. You will then see a full green sunburst.

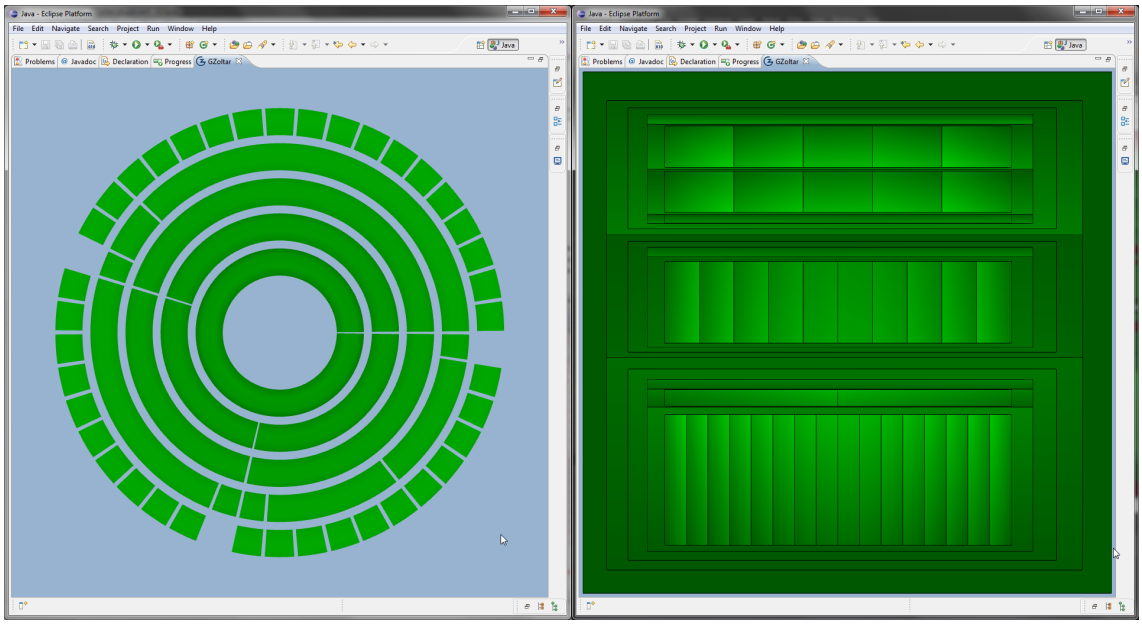

Figure B.20: Fault correction - Step 6

B.4.5 View Location

Your GZoltar view works as any other Eclipse view. Therefore, you can place it wherever you want and resize it to gather your favorite development environment.
We will present some example of different configurations next.

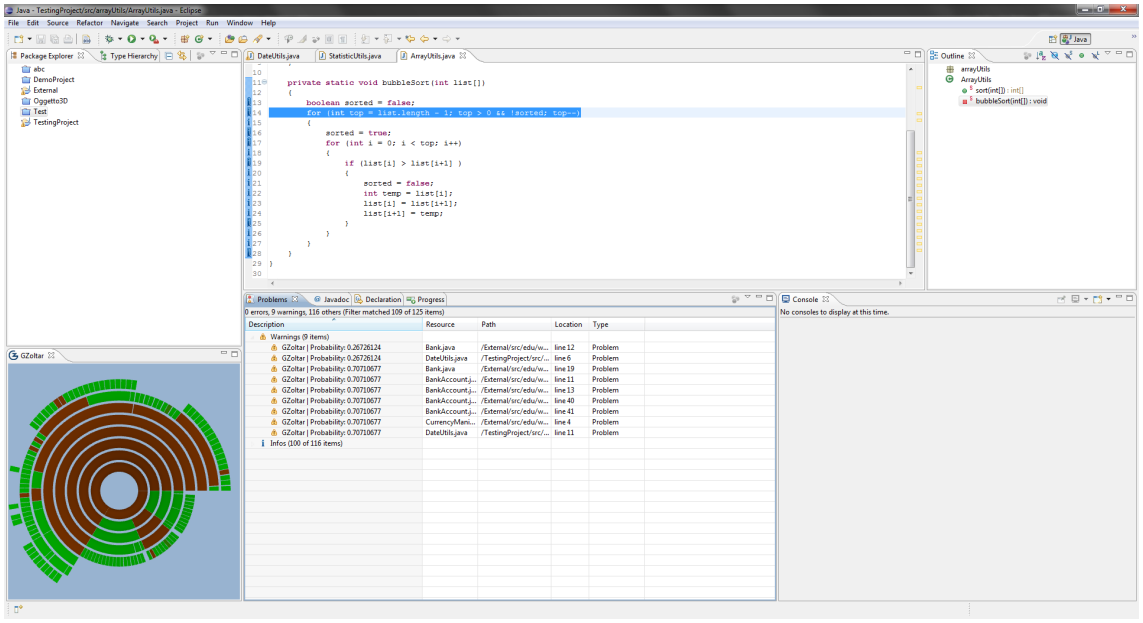

Figure B.21: Sunburst view location - Example 1

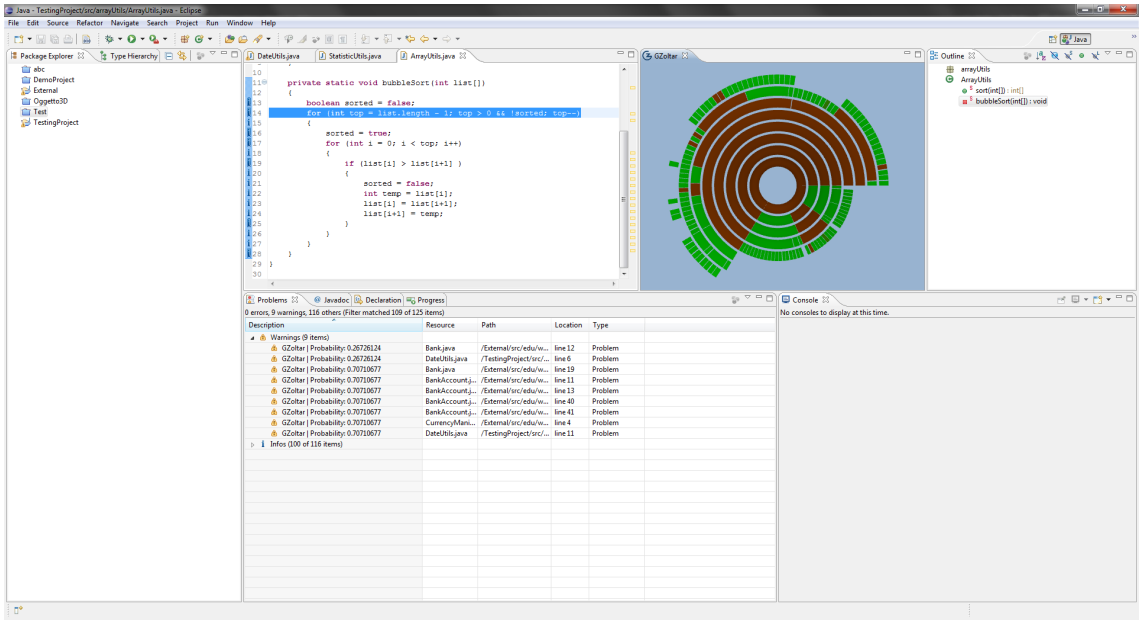

Figure B.22: Sunburst view location - Example 2

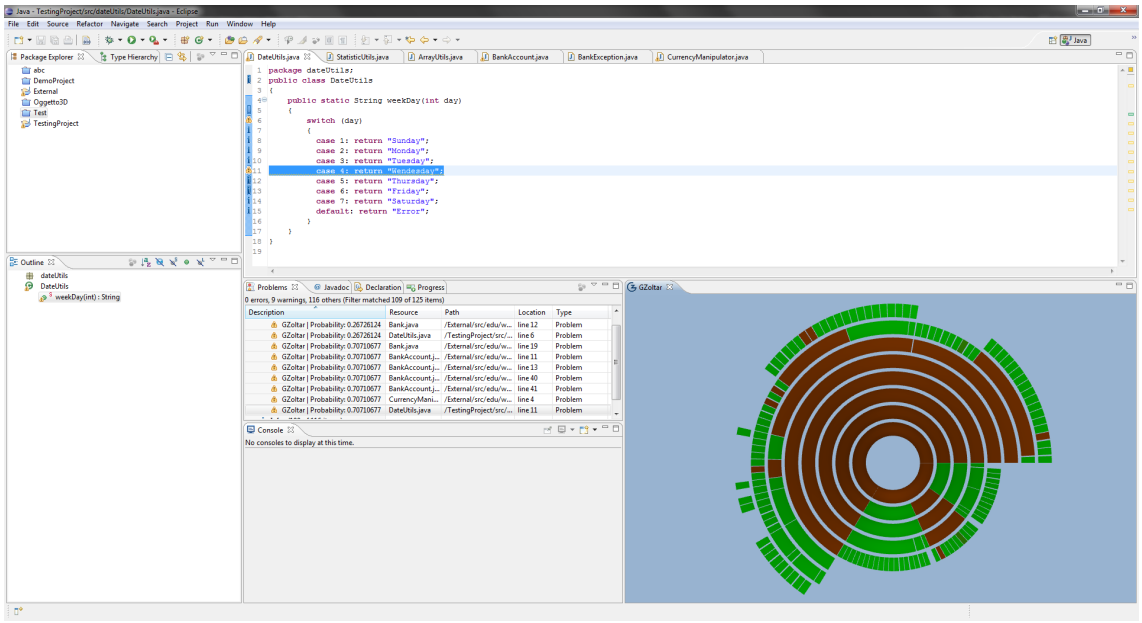

Figure B.23: Sunburst view location - Example 3

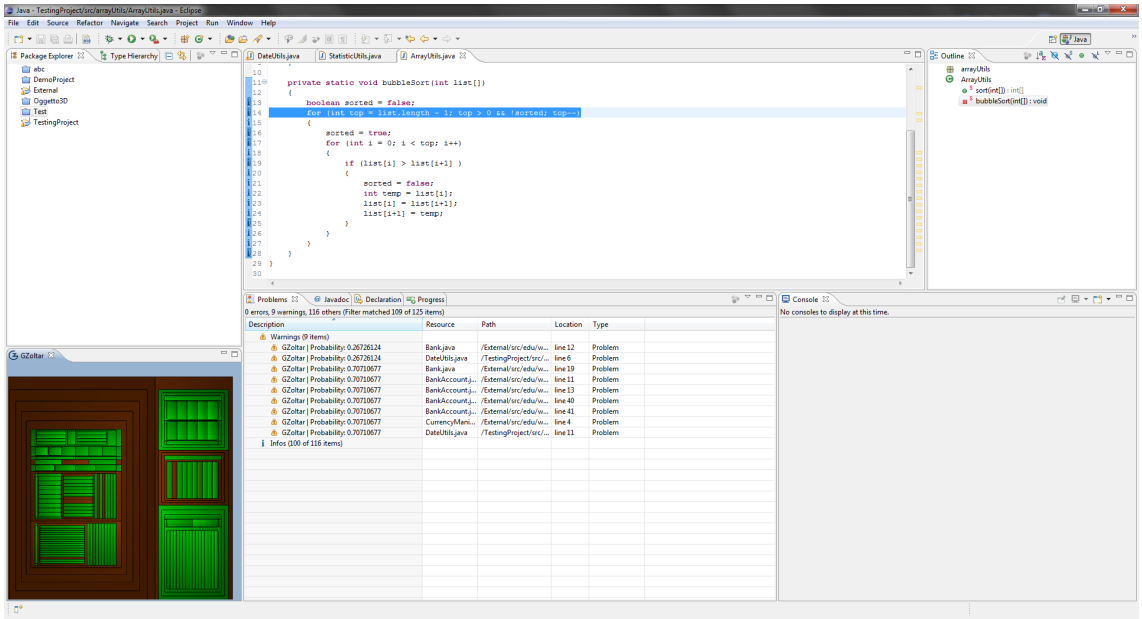

Figure B.24: Treemap view location - Example 1

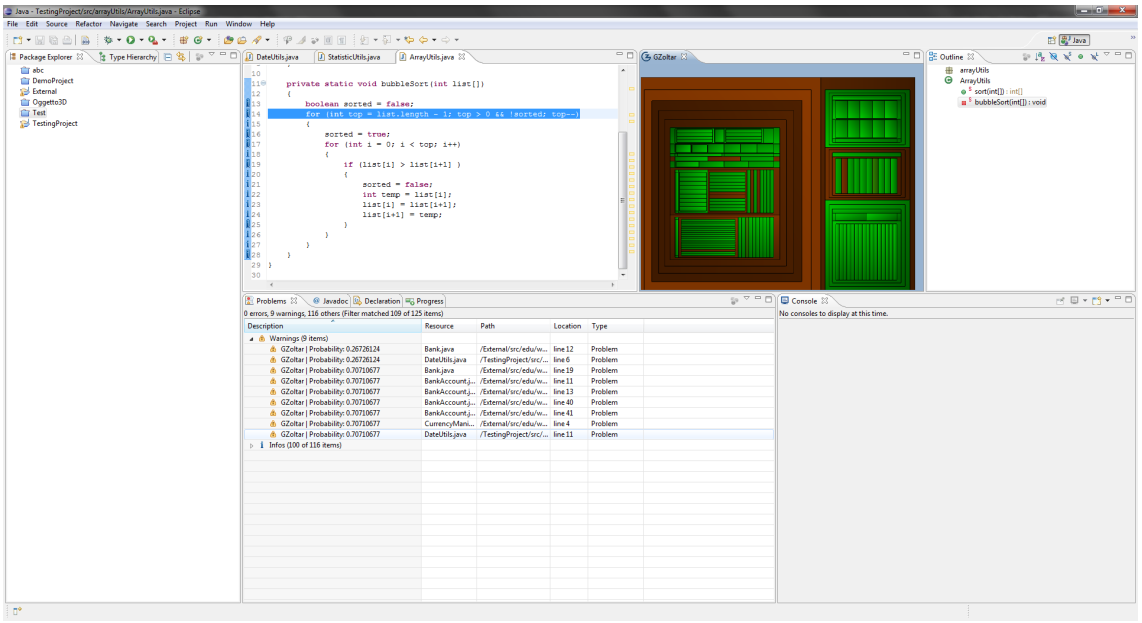

Figure B.25: Treemap view location - Example 2

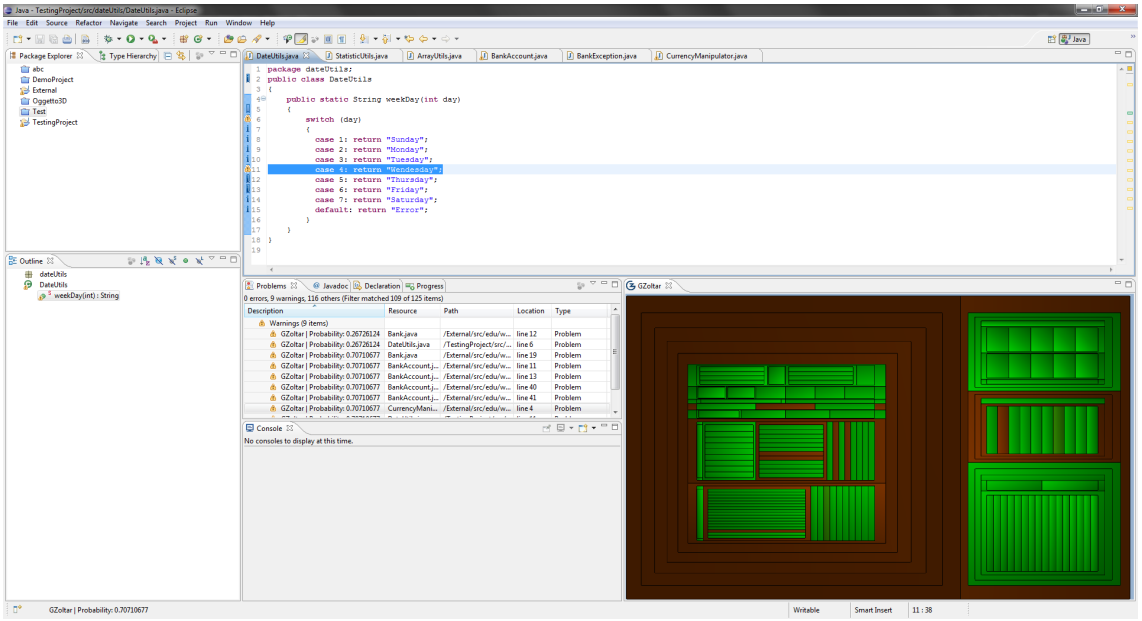

Figure B.26: Treemap view location - Example 3

B.4.6 Multi-Platform

GZoltar is multi-platform. You can use it on Windows (32 and 64 bits), Mac OS X (32 and 64 bits) and Linux (also 32 and 64 bits).

We will show now some examples of GZoltar views being presented on three different systems.

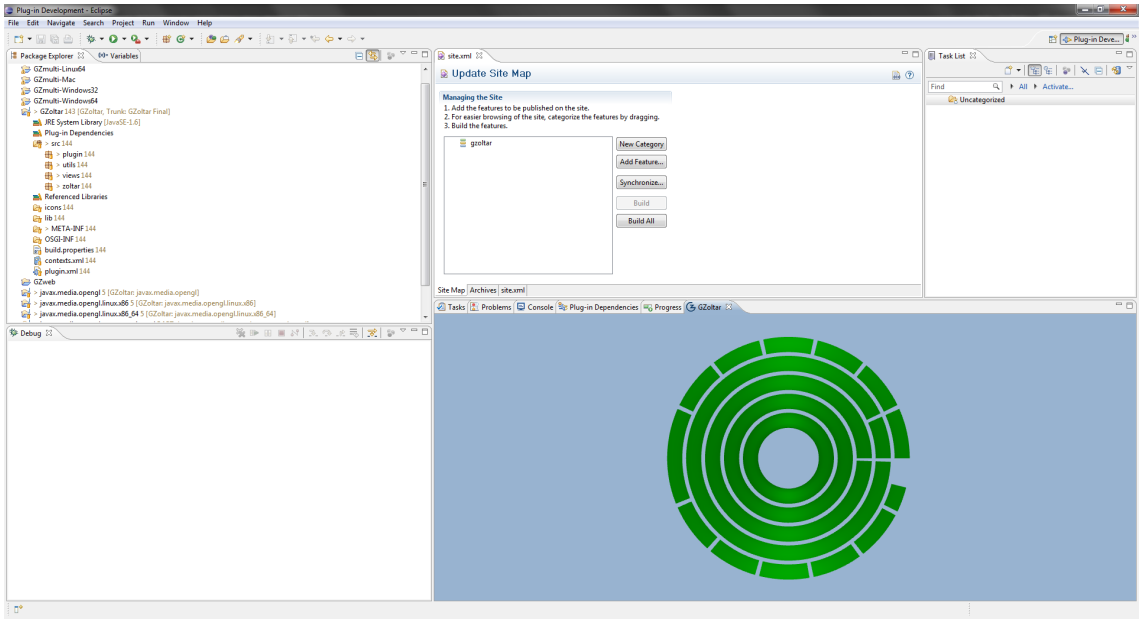

Figure B.27: Sunburst in Windows

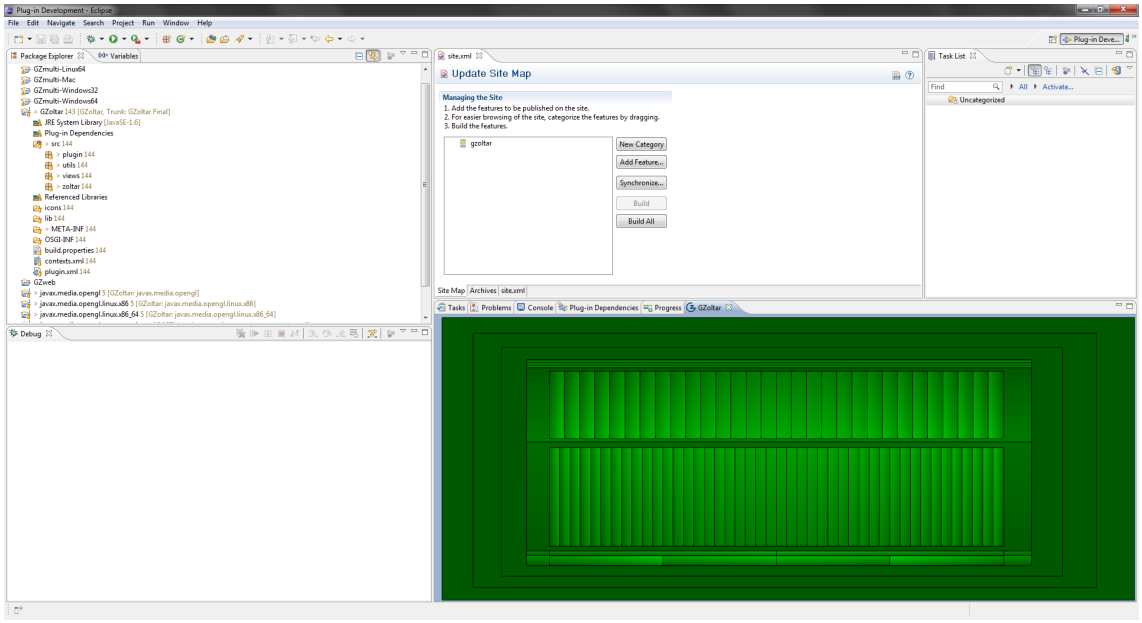

Figure B.28: Treemap in Windows

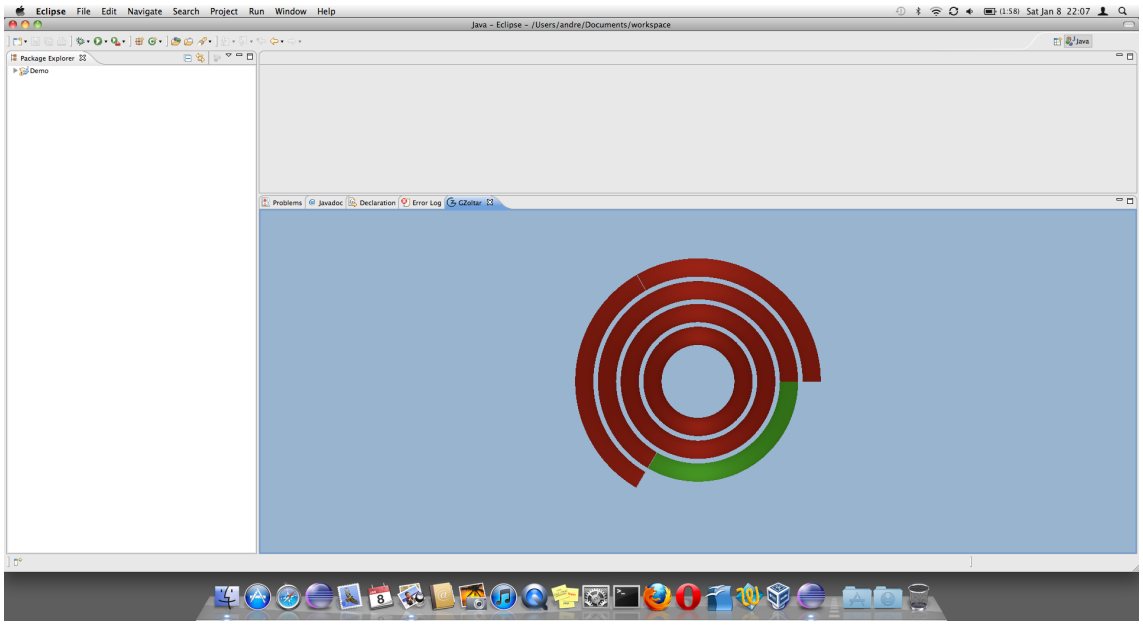

Figure B.29: Sunburst in Mac OS X

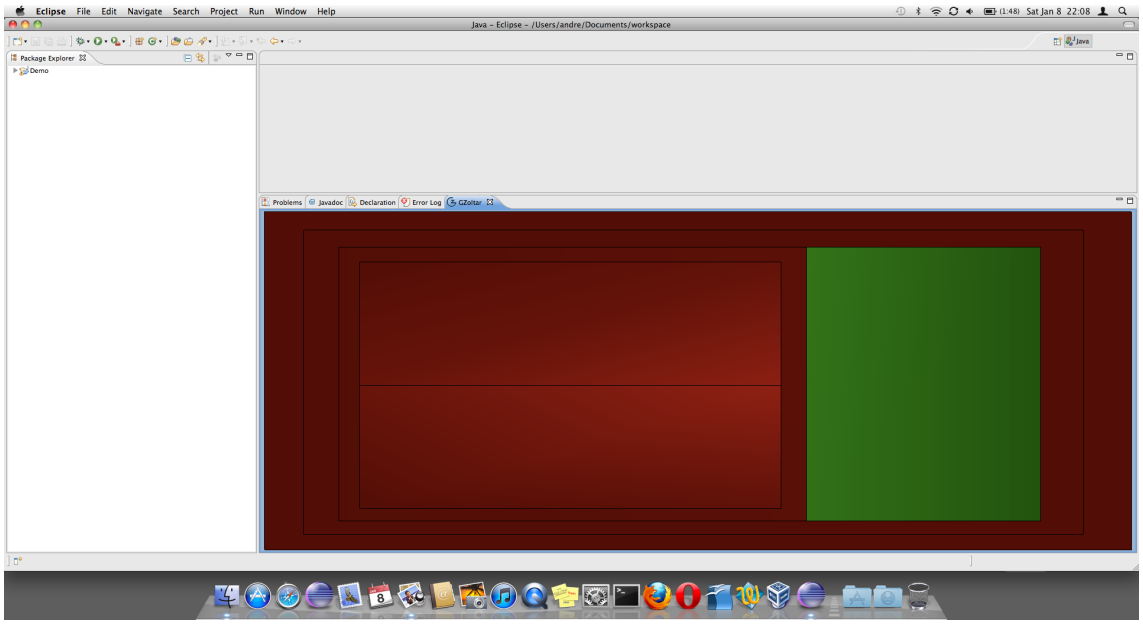

Figure B.30: Treemap in Mac OS X

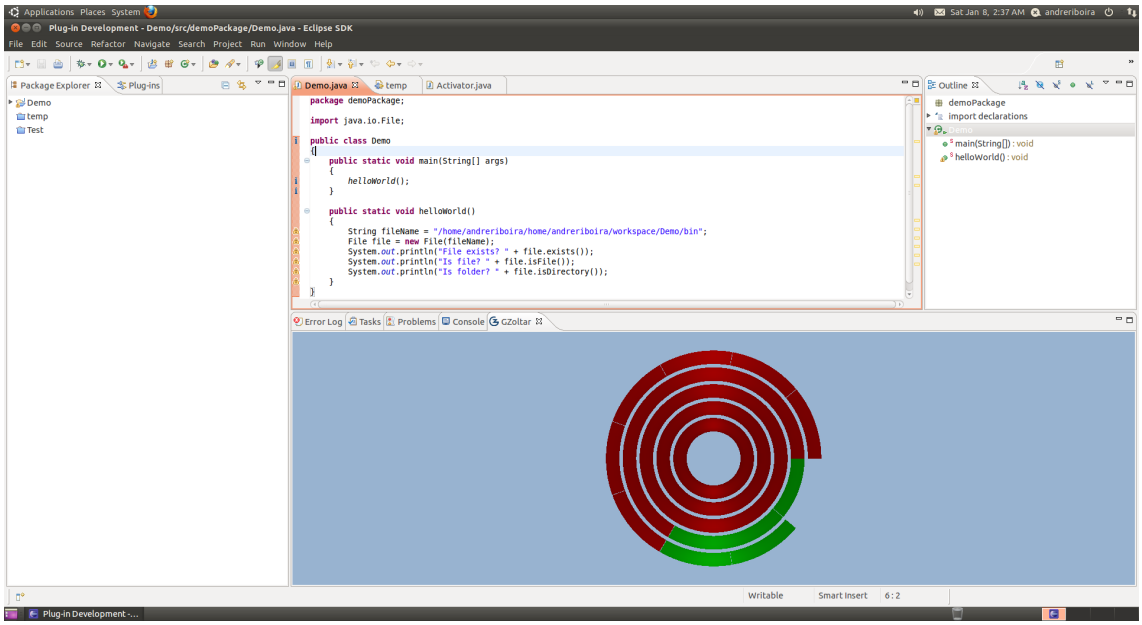

Figure B.31: Sunburst in Linux

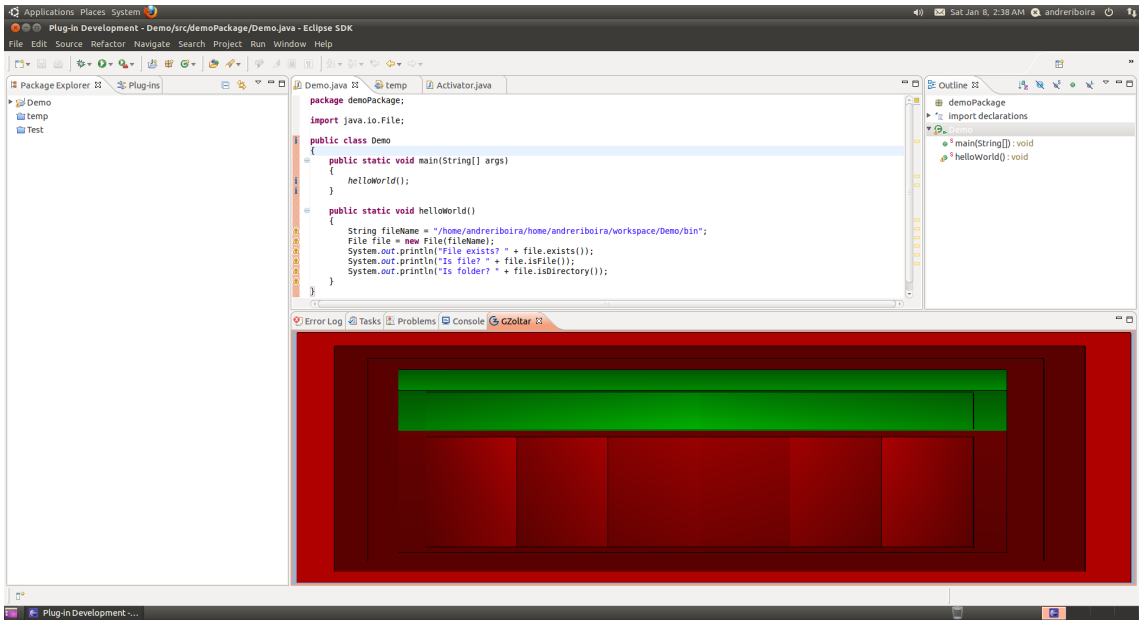

Figure B.32: Treemap in Linux

Appendix C

Timeline

This appendix has GZoltar project timeline.

C.1 Past Work

C.1.1 March 2010

- Rui Maranhão as Zoltar developer stated Zoltar needs for a better visualization.
- André Riboira learned about Zoltar project, about its architecture and goals.
- Rui Rodrigues as computer graphics consultant collected some examples of good graph visualizations.
- "gzoltar" Internet domains were registered, and GZoltar logo was created.

C.1.2 April 2010

- André Riboira learned the technologies involved in this new project, mainly Java programming language, Eclipse IDE architecture and OpenGL.
- André Riboira have created a dummy Eclipse Plug-in that uses OpenGL to create a view. This was only a proof-of-concept to verify the possibility of accept the challenge or not.
- Project websites and Wiki were created.
- Wiki started to be the repository of all the information about this new project, in a private sharing area created to that.

C.1.3 May 2010

- André Riboira studied current visual debugging tools, to learn what is already available and what is missing.
- André Riboira defined some good ways to represent graphs with the needed information for this project (following Rui Rodrigues advices).
- André Riboira made a Multimedia Presentation about GZoltar, highlighting its main goals and the current needs that justify its implementation.

C.1.4 June 2010

- André Riboira and Rui Maranhão compiled some of the research results and submit a position paper about GZoltar to the TAIC-PART 2010.
- Paper was accepted!

C.1.5 July 2010

- Started to test tools and technologies that was planned to be used and integrated.

C.1.6 September 2010

- Presentation of the position paper at TAIC-PART [TAI10] 2010, in Windsor, United Kingdom.
- Started GZoltar development and external tools integration.
- Developed automatic Eclipse projects and classes detection.
- Implemented JaCoCo to instrument and analyze (code coverage) detected classes.
- Ported Ochiai to Java technology.
- Starting thesis writing.

C.1.7 October 2010

- Created code lines relations matrix based on code coverage matrix.
- Developed hierarchical structure tree (processing inner nodes).
- Defined and created object to store all nodes information.
- Continuation of thesis writing.

C.1.8 November 2010

- Developed visualization and navigation framework.
- Continuation of thesis writing.

C.1.9 December 2010

- Created treemap and sunburst views.
- Continuation of thesis writing.

C.1.10 January 2011

- Application testing.
- Final Documentation versions.

C.2 Future Work

- André Riboira's Master of Science (MSc) thesis Presentation at FEUP.
- Write some conference papers with the results we get about this project.