# Integrating Interactive Visualizations of Automatic Debugging Techniques on an Integrated Development Environment

André Riboira, Department of Informatics Engineering, University of Porto, Porto, Portugal & HASLab / INESC TEC, Portugal

Rui Rodrigues, Department of Informatics Engineering, University of Porto, Porto, Portugal & INESC TEC, Porto, Portugal

Rui Abreu, Department of Informatics Engineering, University of Porto, Porto, Portugal & HASLab / INESC TEC, Portugal

José Campos, Department of Informatics Engineering, University of Porto, Porto, Portugal

## ABSTRACT

Automated debugging techniques based on statistical analysis of historical test executions data have recently received considerable attention due to their diagnostic capabilities. However, the tools that materialize such techniques suffer from a common, rather important shortcoming: the lack of effective diagnostic reports' visualizations. This limitation prevents the wide adoption of such tools, as it is difficult to understand the diagnostic reports yielded by them. To fill this gap, the authors propose a framework for integrating interactive visualizations of automatic debugging reports in a popular development environment (namely, the Eclipse integrated development environment). The framework, coined GZOLTAR, provides several important features to aid the developer's efficiency to find the root cause of observed failures quickly, such as direct links to the source code editor. Furthermore, the authors report on the results of a user study conducted to assess GZOLTAR's effectiveness.

Keywords: Diagnostic Reports, Information Systems, Interaction, Statistical Debugging, Visualization

DOI: 10.4018/jcicg.2012070104

## INTRODUCTION

When unexpected behavior is observed on a software system, developers need to identify the root cause that makes the system deviate from its intended behavior. This task (also known as software debugging, fault localization, or fault diagnosis) is the most time-intensive and expensive phase of the software development cycle (Hailpern & Santhanam, 2002). Regardless, it is performed since the beginning of computer history. As an indication of the downtime, debugging, and repair costs involved, a 2002 landmark study indicated that software bugs posed an annual \$60 billion cost to the US economy alone (RTI, 2002). Software faults can also lead to major accidents when occurring on safety-critical systems (Dale & Anderson, 2009). Testing and debugging tasks should therefore receive considerable attention during the software development life-cycle. Hence, it is important to have powerful tools to help testers and developers on that crucial task.

Visualization is rather important for information comprehension (Van Wijk, 2005). In general, human beings find it more intuitive to understand information laid out in a logical, hierarchical way, than with a simple, rather plain and textual list of values. Nevertheless, currently available automatic debugging tools do not offer powerful visualizations of their debugging reports (Riboira, 2011).

In this paper, we present GZOLTAR, a visual debugger for Java programs that aims to fill the gap of currently available automatic debugging tools. The main premises for GZOLTAR are therefore the following:

- Implement a robust automatic debugging framework that allows different visualization techniques, and that may be easily expanded in the future;
- Help the user to find software faults faster, by aiding the understanding of debugging results;
- Be highly integrated in a multi-platform development environment to reduce the

learning curve, and the time spent on swapping between faults' localization and their fixing;

Have an easy and fast installation process to facilitate its adoption and use;

The automatic debugging tool behind GZOLTAR is ZOLTAR (Janssen, Abreu, & Van Gemund, 2009), a Spectrum-Based Fault Localization (SFL) framework whose performance is amongst the best ones for fault localization (Abreu, Zoeteweij, & Germund, 2009). GZOLTAR is implemented as a plug-in for the Eclipse integrated development environment (IDE) (Burnette, 2005) due to its wide adoption (Geer, 2005) and plug-in development facilities (McCullough, 2006). The interactive visualization framework uses OpenGL for graphics rendering due to (i) its flexibility to produce both 2D and 3D graphics, (ii) its performance supported by hardware acceleration, and (iii) its multi-platform availability (Shreiner & Group, 2009). As Eclipse cannot access OpenGL directly, some supporting libraries such as JOGL were used to create bindings to OpenGL native system libraries (Wolf, 2005). The interactive visualization framework can be easily extended with new visualizations, and we present two examples of possible visualizations as proofof-concept: sunburst and treemap (Stasko, Catrambone, Guzdial, & McDonald, 2000).

The proposed interactive visualization framework eases the creation of different debugging data visualizations and allows navigation and integration with default Eclipse features, such as the code editor and the building warnings' list.

The remainder of this paper is organized as follows. In the next section we present a review of the state-of-the-art. Then, we explain automatic debugging and outline the ZOLTAR framework. The presentation of the GZOLTAR's architecture follows. Subsequently, we propose GZOLTAR's visualizations, followed by their interaction. Next, we report on the results of the user study. Finally, we conclude and discuss future work.

#### State-of-the-Art

A traditional approach to fault localization is to insert "print" statements in the program to generate additional debugging information to help identifying the root cause of the observed failure(s). Essentially, the developer adds these statements to the program to get a glimpse of the runtime state or to verify whether the execution has reached a particular part of the program or not. Another common technique is the use of a symbolic debugger, which supports additional features such as breakpoints, single stepping, and state modifying. Examples of symbolic debuggers are GDB (Stallman, 1994), DBX (Sun Microsysterms, Inc., 1990), DDD (Zeller & Lutkehaus, 1996), EXDAMS (Balzer, 1969), and the debugger proposed by Agrawal et al. (1991). Symbolic debuggers are included in many IDE such as Eclipse<sup>1</sup>, Microsoft Visual Studio<sup>2</sup>, Xcode<sup>3</sup>, and Delphi<sup>4</sup>. Whyline (Ko & Myers, 2009) is a graphical debugging tool that uses a different concept, where the user is able to query "why" the software behaves in a certain way. Whyline is more oriented to the dynamic execution path (i.e., dynamic slicing), and not so much oriented to the structure of the source code. It uses dynamic slicing to track down the set of statements that have an impact in a specific value. Furthermore, it is not designed to calculate the failure probability based on the results of a set of test executions, but only to discover statements that impact another one. The user has to interact almost permanently with the tool, and analyze each failing execution of the system under test (SUT).

These traditional, manual fault localization approaches have a number of important limitations. The placement of print statements as well as the inspection of their output are unstructured and ad-hoc, and are typically based on the developer's intuition. In addition, developers tend to use only test cases that reveal the failure (i.e., failing test cases), and therefore do not use valuable information from successful test cases. Furthermore, the size of the program state at each point can be large, and there are many combinations of program executions that have to be examined. Hence, such techniques still require a detailed knowledge of the program, and also suffer from a substantial execution overhead in terms of execution time and space to store historical run-time data. Last, but not least, manual debugging is extremely expensive in terms of labor cost.

Aimed at drastic cost reduction, much research has been performed in developing automatic fault localization techniques/tools, being the spectrum-based fault localization (SFL) techniques the most efficient (Janssen, Abreu, & Van Gemund, 2009). Notwithstanding, little work has focused on developing a visual representation of the diagnostic report, being EZUnit (Bouillon, Krinke, Meyer, & Steimann, 2007) and Tarantula (Jones, Harrold, & Stasko, 2002) the most significant efforts. EZUnit is an Eclipse plugin that presents a list of possible faulty lines of code, based on statistical analysis of the system under test, but does not implement SFL techniques. EZUnit can also present a call-graph of the system under test. Tarantula is a standalone tool that exhibits a representation of the source code of the system under test, highlighted with each line's failure probability. Tarantula's visualization resembles a code editor seen with a huge zoom out factor, to allow the user to have a picture of the entire project. EZUnit and Tarantula do not offer a way to visualize the structure of the SUT, neither how lines of code relate between each other in terms of simultaneous executions

#### AUTOMATIC DEBUGGING USING THE ZOLTAR FRAMEWORK

The process of pinpointing the fault(s) that led to symptoms (failures/errors) is called fault localization, and has been an active area of research for the past decades. Based on a set of observations, automatic approaches to software fault localization yield a list of likely fault locations. This list is subsequently used either by the developer, to focus the software debugging process, or by automatic recovery mechanisms (Patterson et al., 2002).

GZOLTAR's automatic debugging core, ZOLTAR, is a recent framework that performs statistical analysis of historical software test executions' data to calculate the failure probability of each component of a SUT. That calculation is made using SFL algorithms. ZOLTAR'S SFL algorithms are among the most efficient (Abreu, Zoeteweij, & Germund, 2009). They are being successfully used on academic and industrial environments, and are currently under active development.

#### SFL Concepts

The most predominant approaches to fault localization can be classified as (1) statistical approaches or (2) reasoning approaches, depending on the amount of knowledge that is required about the system's internal component structure and behavior. The former approach uses an abstraction of program traces, dynamically collected at runtime, to produce a list of likely candidates to be at fault, whereas the latter combines a static model of the expected behavior with a set of observations to compute the diagnostic report.

A statistical approach to spectrum-based fault localization (SFL) will now be described. A program under analysis comprises a set of M components (e.g., functions, statements)  $c_j$  where  $j \in \{1, ..., M\}$  and can have multiple faults, the number being denoted C (fault cardinality). A diagnostic report  $D = <..., d_{k'} ... >$ 

is an ordered set of diagnosis candidates d ordered in terms of likelihood to be the true diagnosis. Statistical approaches yield a singlefault diagnostic report with the M components ordered in terms of statistical similarity (e.g., <{3}, {1}, ...>, in terms of the indices j of the components c). Program (component) activity is recorded in terms of program spectra (Harrold, Rothermel, Wu, & Yi, 1998). This data is collected at runtime, and typically consists of a number of counters or different components of a program. In this paper we use the so-called hit spectra, which indicate whether a component was involved in a (test) run or not. Both spectra and program pass/fail (test) information are input to the SFL algorithm. The program spectra (see Figure 1) are expressed in terms of an  $N \times M$  activity matrix A. An element a<sub>ii</sub> is equal to 1 if component j was observed to be involved in the execution of run i, and  $\theta$  otherwise. For  $j \leq M$ , the row A<sub>i</sub> indicates whether a component was executed in run i, whereas the column  $A_{*_i}$  indicates in which runs component j was involved. The pass/fail information is stored in a vector e, the error vector, where e, represents whether run *i* has passed  $(e_i=0)$  or failed  $(e_i=1)$ . Note that the pair (A, e) is the only input to SFL.

In SFL one measures the statistical similarity between the error vector e and the activity profile column  $A_{*j}$  for each component  $c_j$ . This similarity is quantified by a similarity coeffi-

Figure 1. Matrix A (SFL input matrix). N means test executions, M means SUT components, a means code coverage and e means test execution result



cient, expressed in terms of four counters  $n_{pq}(j)$  that count the number of elements in which  $A_{*j}$  and *e* contain respective values *p* and *q*, i.e, for *p*,  $q \in \{0,1\}$ , we define  $n_{pq}(j) = |\langle i | a_{ij} = p \land e_i = q \rangle|$ . An example of a well-known similarity coefficient is the Ochiai coefficient, which is among the best for fault localization [1,24]

$$s(j) = \frac{n_{_{11}}(j)}{\sqrt{\left(n_{_{11}}(j) + n_{_{I\theta}}(j)\right) \cdot \left(n_{_{11}}(j) + n_{_{\theta I}}(j)\right)}}$$

For detailed information, refer to Abreu, Zoeteweij, and Germund (2009). Ochiai is also implemented in the ZOLTAR toolset (Janssen, Abreu, & Van Gemund, 2009).

To compute the failure probability of software components – typically lines of code – ZOLTAR requires information about the number of times a given component was involved in failed and successful tests. For this it is necessary to instrument the source code, so that during execution the information of which lines were executed needs to get recorded. This record represents the execution's code coverage. With this information and the result of the test execution (if it passed or failed), ZOLTAR is able to calculate the failure probability of each system component. This input data is received by ZOLTAR in the form of a code cov-

erage matrix, where each column represents a system component, and each line represents a test execution. The result of the test executions is received by ZOLTAR as an error vector (see Figure 1) (Abreu, 2009).

Because ZOLTAR is based on test executions' code coverage and results, its accuracy will depend on the quality of test cases. During the statistical analysis, ZOLTAR processes the failure probability of each system component, and at the end it returns that information in the form of a list, with the component and its corresponding failure probability.

ZOLTAR's core automatic debugging processing is very efficient but this tool has some shortcomings, mainly related to its user interface. Its default output is a list with the system components and their failure probability (see Figure 2), presented in a text-based user interface. ZOLTAR also has a graphical interface, XZOLTAR, but it is very limited. XZOLTAR is essentially a code viewer with each line of code highlighted and color-coded with its failure probability (see Figure 3). The lack of integration with an IDE is also an issue, because the developer has to localize its faults on one environment and fix them on another, which may lead to a loss of productivity. Besides, at this moment XZOLTAR is only available for the Linux operating system.

Figure 2. Zoltar output

	Software Analyzer v0.2.4 Delft University of Technology		
Operating Mode Runs -Spectra	SFL analysis		
Invariants	spectrum name Basic	Blocks	
Exit	number of components 35		
	SFL coefficient Ochiai		
	rank         score         component info           0         0.577350         textVal.c:43 -           1         0.500000         textVal.c:29 -           2         0.447214         textVal.c:19 -           3         0.447214         textVal.c:19 -           5         0.447214         textVal.c:20 -           6         0.447214         textVal.c:22 -           7         0.447214         textVal.c:26 -           8         0.447214         textVal.c:28 -           9         0.447214         textVal.c:33 -		
	pgup , pgdn—^v	backspace_	





#### GZOLTAR INTERACTIVE VISUALIZATION'S FRAMEWORK ARCHITECTURE

GZOLTAR is an Eclipse Plugin (McCullough, 2006) developed in Java. It accesses Eclipse's Workspace features to be able to obtain information about opened projects and it accesses Workbench features to be able to create the plugin's user interface.

GZOLTAR components can be divided into five main areas (for a representation of GZoltar technological layers see Figure 5):

- 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 produces code coverage information, to create the needed SFL matrix. ZOLTAR executes the Ochiai algorithm 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 Figure 4.

Because GZOLTAR's output is an Eclipse view, it uses the toolkit that produces Eclipse views SWT which is part of Eclipse's Workbench (Lintern, Michaud, Storey, & Wu, 2003). Inside the view, GZOLTAR uses OpenGL as the base technology for rendering. OpenGL's multi-platform support and hardware acceleration support, when properly used, allow to efficiently display complex scenes exploring the potential of 2D and 3D graphics. Furthermore, its well-known API eases the learning curve for someone who wants to create new visualizations within GZOLTAR (FIGURE 5). Although OpenGL is not available directly for Java, there is a tool, JOGL (Wolf, 2005), that provides OpenGL library bindings to Java.

Figure 4. 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 the default Code Editor



To render OpenGL scenes, JOGL uses Java's AWT (O'Neal & Stewart, 1996), which is a multiplatform toolkit to produce Java GUI's (Lintern, Michaud, Storey, & Wu, 2003). Eclipse has a bridge that connects SWT and AWT, allowing Eclipse to have OpenGL scenes on one of its default views. Finally, GZOLTAR needs to obtain the code coverage info (if a component was used or not) about each test execution, to be able to calculate the failure probability of each component. To obtain that info, GZOLTAR uses JaCoCo (Hoffmann, 2011).

Figure 5. GZoltar technological layers



Creating a new visualization for GZOLTAR is a simple task, because the visualization framework provides a set of features that allows the developer to focus only on the OpenGL scene. The developer has access to a tree-structured data set, which has all the information needed for the visualization itself. Some features are common to all visualizations, and do not need to be implemented by the developer. For instance, the node color and label are automatically set by GZOLTAR's visualization framework. Other features such as the nodes' highlighting and selection (to expand or jump to the source code), the source code lines relationship highlighting, and navigation features like zoom and rootchange are also provided by the visualization framework. The visualization developer only needs to create a cycle that renders all nodes from the received tree-structured data set, with the desired shape and location. Every other feature is assured by the visualization framework. It is the framework that determines which nodes will be displayed and when, in which color, and with which label, according to the debugging report results and the user interaction for data navigation. Nevertheless, the visualization developer is able to create powerful 3D visualizations, and if wanted, can even bypass some of the automatic features provided by the visualization framework, to have more control over the scene.

As an example, a pseudo-code is presented in Box 1, with a sample visualization that could be added to GZOLTAR's framework:

#### VISUALIZATIONS OF DEBUGGING RESULTS

As stated earlier, current automatic debugging tools (including ZOLTAR) lack efficient visualizations and integration with IDE's. The purpose of GZOLTAR is to fill this gap, by presenting a visualization of the SUT's source code structure. The visualization is complemented with debugging information related with each SUT component.

Eclipse uses the workspace paradigm. A workspace contains a set of projects, each consisting of packages with classes within, and ultimately lines of code, which correspond to the components of the SUT in GZOLTAR's terminology. This structure can be represented as a tree where each node represents a component or a group of components. A node can therefore be a project, a package, a class, a method or a line of code (these being the leaves of the tree). GZOLTAR supports multi-level packages, so it has to deal with trees with an arbitrary number of levels.

There are many ways to visually represent tree data structures. A given representation may be effective for a tree with a given pattern of node weight distribution and node relationships, but too confuse for a differently structured tree. Furthermore, people react differently to the same visualization, so it is important to give alternatives to the user (Stasko, Catrambone, Guzdial, & McDonald, 2000).

```
Box 1.
public class NewVisualization {
    public static void draw(GL gl, float ratio, ZoltarTree z) {
        Vector<ZoltarTree> children = z.getChildren();
        gl.(...); // Scene's OpenGL code goes here
        for(ZoltarTree child: children) {
            String curNode = child.getLabel();
            String curRoot = Visualizations.getCurRoot();
            Float curPercent = z.getPercent();
            Float[] colors = Visualizations.nodeColor(child,
false);
            gl.(...); // Node's OpenGL code goes here
            }
        }
}
```

For this reason, GZOLTAR was implemented in a modular way that easily allows adding different visualizations to the system. In the current version of the tool, two visualizations are provided: sunburst and treemap. The first is more focused on the tree hierarchy, while the other is more focused on the tree leaves. In both of them, the debugging information, namely the component failure probability, is represented by color-coding each node, using colors ranging from pure green (no failure probability) to pure red (maximum failure probability). The two visualizations are described in more detail in the following sections.

#### Sunburst

Sunburst is a circular visualization (Stasko, Catrambone, Guzdial, & McDonald, 2000) that can be compared to a multiple-level ring graph. Each level of the visualization represents a different hierarchical level (packages, classes, methods, etc.) of the tree-structured data. Because it supports multi-level packages, the same level of the visualization can represent different kinds of components (the same visualization level can have classes and packages, for instance). The tree leaves (which represent lines of code) have a fixed area, calculated by the total number of lines of code on the system. The inner nodes (methods, classes, etc.) have their area calculated based on the sum of the areas of their descendants (see Figure 6). Sunburst uses the green-to-red coloring scheme referred earlier. It has however an additional coloration method that is activated on user interaction: when the user places the mouse cursor over a representation of a line of code, the coloration of the visualization changes to reveal the relations between different lines of code of the system.

When the user places the mouse cursor over a leaf node, all the inner nodes will render in gray color, and the leaf nodes will render in a color that varies from the color of the selected node to gray, depending on the relationship between that node and the selected one. This information is obtained from the percentage of simultaneous executions, between two given lines of code. With this information it is possible not only to have a notion about the way components relate with each other, but also the depth of the relation.

#### Treemap

Treemap is a rectangular visualization (Johnson & Shneiderman, 1991) that is widely used on disc space usage analyzers, because it focuses more on the tree's leaves than on its hierarchy. Each node is represented as a rectangle with an inner margin, and its interior is divided proportionally by its descendants according to their weight. To avoid node representations with disproportionate width and height, nodes

Figure 6. GZoltar sunburst visualization



at odd levels of the hierarchy are divided horizontally and nodes at even levels are divided vertically (see Figure 7). With this concept the leaves occupy the majority of the display area (the remaining corresponds to the margins).

#### Comparison

These two visualization concepts highlight different aspects so they are both useful. Sunburst focuses more on the tree hierarchy, which reflects the system organization. The system organization knowledge is important to isolate groups (packages, classes, etc.), which should be seen in more detail (in this case, which have higher failure probability). Treemap focuses more on the tree leaves, which represent lines of code. Fast access to lines of code is important when errors are well isolated, and the user wants to access directly to the source code at the desired line. Furthermore, as the render area is rectangular and treemap offers a rectangular visualization, it makes better use of the available space.

A comparison between Sunburst and Treemap view is displayed in Figure 8. GZOLTAR considers all packages as 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's structure, to help the user in his fault localization task.

To better understand the differences between visualizations, two sample systems are presented in Figure 9, having sunburst and treemap visualizations side-by-side. It is clear that although the visualizations provide an overview of the systems and the fault probability distribution, the more complex system is not trivial to analyze using just this broad view. In these cases, it would be useful to have additional control over the visualization, allowing to focus the visualization on specific parts of the system, and even accessing the faulty sources directly from the visualization. This leads to the other important component of GZOLTAR, the interaction with the visualization and its connection to the IDE, presented in the next section.

## INTERACTION

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 help the developer during software development. Those tools can give not only useful functionalities about code editing, like line numbers and syntax highlighting, but also about project organization, code completion, integrated help and the ability to analyze the system state at a given stage. However, the most state-of-the-art automatic debugging tools (e.g. Tarantula and ZOLTAR) are external to

Figure 7. GZoltar treemap visualization





Figure 8. GZoltar visualizations comparative

IDE's, which may compromise their adoption. To ease wide adoption, GZOLTAR is integrated in the Eclipse IDE. All GZOLTAR's visualizations are rendered on a standard Eclipse view. This allows the user to place and resize the visualization area to the desired place and size, to enhance comfort. Default Eclipse code editors can be opened directly from the visualization, and standard Eclipse warnings are generated by GZOLTAR. Those warnings are displayed on Eclipse "Problems" list, and as tooltips in the code editor (see Figure 10; for a video demonstration of GZOLTAR, please consult (Riboira, 2011)).

A user can interact with GZOLTAR visualizations using a mouse and a keyboard. The user can expand and collapse progressively each of the system components or expand all components at a time. Zooming and panning is also possible to increase detail in a particular visualization area. It is also possible to make a "root change", by choosing any inner tree node to be the new visualization root. The user can also swap between visualizations.

#### Navigation

By default, only the components placed on the top tree level are displayed. The user can expand any inner tree node by clicking on it to navigate through the project structure (see Figure 11). If the user clicks on a node that is already open, it will collapse. Pressing the "space" key will expand all nodes. If the user presses the "space" key again it will return to the previous state. Navigation history is preserved even on visualization swap. When the user clicks on a representation of a line of code, an Eclipse

Figure 9. GZoltar visualizations of a simple (top) and complex (bottom) systems





Figure 10. GZoltar integration with Eclipse IDE

code editor is opened with the corresponding source file. The text cursor is placed on the corresponding line of code so the developer can quickly fix the fault.

#### Zoom and Pan

One feature that is useful to the user when dealing with large systems is zoom and pan. A user can zoom into a specific visualization area using the mouse wheel, the keyboard or by doubleclicking on a visualization spot without releasing the mouse button on second click. Panning is also possible by clicking on a visualization spot and dragging the visualization without releasing the left mouse button. By zooming in and panning to the desired place, the user can click easily on tiny nodes, or analyze in detail a small portion of the system (see Figure 12).

#### **Root Change**

Another feature useful for large projects is root change. The user can chose any inner tree node

Figure 11. GZoltar navigation

Steve - Eckpoe		a - Edipte	H_HILF
File Edit Nevigete Search Project Run Window Help	The later	Edit Navigete Search Project Bun Window Help	
	Presert T	- M H M H	E Mara C Resource
2 More II and II Balano (Papar (3.306 ct)	Fallure: 57.735027%	Testing <sup>b</sup> rojectistuisticUllis/StatisticUllis.java averageB0 (Fal	ure: 67.796027%
D.	A 4 5 01		0 11 1

Figure 12. GZoltar zoom and pan



to be the new visualization root. To do a root change, the user has to click on a node with the right mouse button. A new visualization is then created, with the same levels, but ignoring all the nodes that are not directly related to the chosen one. Only its ancestors, descendants and siblings will be present on the new visualization (see Figure 13). To return to a more complete tree, the user just has to click on a node that belongs to a parent level. Selecting a tree node of the first level to be the new visualization root will display the entire tree.

## USER STUDY

In order to validate the usefulness of the current version of the plug-in, seven users were selected to test the efficiency of the interactive visualizations. It was recorded the time that each user took to finish a debugging task. At the end of this process, each user filled a form with the feedback of their experience and some suggestions for future work. This usability test was important not only to test the efficiency of the presented plug-in but also to aid the development team to fulfill the user's needs in future versions of this tool.

#### **Users Description**

Seven developers composed the users group. The users were picked randomly from the Department of Informatics Engineering of the Faculty of Engineering at the University of Porto.

The number of users was based on J. Nielsen's work related with usability and user tests (Nielsen & Landauer, 1993). Nielsen advocates that for a small software project, seven is the optimal number of users to participate in the usability test (Nielsen & Landauer, 1993). This small number of users should be enough to identify the main usability issues. This experiment was conducted to identify the main users' difficulties while using the GZOLTAR plug-in. This information was helpful to create guidelines to improve future versions of this tool and to have a first assessment of the impact of this plug-in among the users.

The user group was composed of MSc and PhD students in Informatics Engineering, aged 22 to 26 years old, and from both genders. The users were familiarized with three main operating systems: Linux (85.7%), Microsoft Windows (85.7%) and Apple Mac OS X (42.9%). The most common programming languages used by the group members were Java (100%), C (100%), C++ (100%), PHP (100%), C# (85.7%), Python (71.4%) and Assembly (57.1%). The majority of the developers used regularly an IDE, being the most popular IDEs Eclipse (85.7%) and Microsoft Visual Studio (85.7%). However, 14.3% of the users did not use regularly any IDE. The most used debug-



#### Figure 13. GZoltar root change

ging techniques were breakpoints (85.7%) and "print" statements in the code (71.4%). JUnit is the main testing tool used by the group (71.1%). On the other hand, 28.6% of the elements of the group stated that they do not use any particular testing technique.

#### **Experiment Conditions**

The experiment was conducted in laboratories of the Department of Informatics Engineering and classrooms of the Faculty of Engineering at the University of Porto. Computers with Linux and Eclipse IDE with the GZOLTAR plug-in were available for the experiments.

Each user had to debug a faulty version of the NanoXML v2.2.3 application<sup>5</sup>. This application has 5396 lines of code (LOC), and a suite of JUnit tests. A fault was injected in the class XMLUtil, from the net.n3.nanoxml package. The line 109 (from method skipTag) was changed from "case '>':" to "case ']':". Users had no previous contact with the application source code and the JUnit tests. A brief explanation (less than 5 minutes) was given to each user, to explain the goals of the task (pinpoint the faulty code) and how GZOLTAR works.

The users had 20 minutes to localize and fix the faults. After the debugging task, each user filled a survey with questions on their experience.

#### **Results and Feedback**

As mentioned before, the time limit for this task was very short-only 20 minutes. The goal was not to record how long the users would take to find and fix the faults, but to obtain feedback about the plug-in usability and usefulness by a set of independent users. It is important to note that 71.4% of the users were able to find the fault in less than 20 minutes (and 42.9% even fixed the fault to ascertain that the suspicion was justified). It is important to highlight that the users did not know which application was going to be used in the experiment, and they did not have any previous knowledge about the source code. From the users that were not able to find and correct the fault. 50% were able to point the most likely fault localization. However, because they were not able to fix the fault, they could not confirm that the fault localization was right. It is important to note that some users were rather uncomfortable with the Eclipse IDE because they never used it or they did not use it on a regular basis.

The survey had a section where the users answered questions related with their profile and development experience, and a section where the users could give their feedback. Replies to the questions about the plug-in interface, performance and associated concepts were

provided using a scale from 1 (unacceptable) to 5 (excellent).

A considerable amount of users (42.9%) found the plug-in difficult to understand at first. However, the majority (57.1%) stated that they understood in a short period of time how the tool works. The debugging tasks, using GZOLTAR plug-in, were considered fast and logic by 71.4% of the users.

The users group also analyzed the performance of the plug-in. An expressive slice of the group (85.7%) considered the responsiveness of the plug-in as very good. The majority of users (71.4%) found that the plug-in usefulness increased with their experience and knowledge about the tool. Most importantly, all the users considered that they were able to obtain good results with little knowledge about the tool operation.

The users group also gave feedback about the associated concepts in GZOLTAR. All the users considered automatic debugging as an important concept, where 85.7% classified it as "essential". Debugging techniques integrated into IDE were also considered important, having the majority of users (57.1%) considered them as "essential". A large number of users (71.4%) also considered visual debugging as an important concept.

The final part of the survey had an open question where the users could leave their comments and suggestions. Some suggestions were related with the colors. Some users found that the full-color spectrum affected negatively the visualization analysis. They suggested the limitation of the number of colors (having for example color red for "high probability", yellow for "low probability" and green to "no probability"). The users' comments were very positive. Two users stated that without the GZOLTAR plug-in, they would probably never have found the software faults, because they did not know the software they were testing.

This experiment with developers validated our hypothesis. An interactive visualization of automatic debugging reports can help developers to find fault localizations in a short period of time. Moreover, an IDE plug-in facilitates not only the faults localization but also the fixing of the localized faults. Even not knowing the faulty software, most of the participants were able to find and fix the faults in less than 20 minutes.

## CONCLUSION AND FUTURE WORK

Debugging is an important task in software development. Almost all existent software has bugs that can lead to a considerable loss of productivity and money. Debugging software - finding and fixing faults - is a time-intensive task. There are some tools and techniques to help in bug localization. Some of them use statistical approaches to allow automatic debugging. GZOLTAR's goal was to fill a gap in current automatic debugging tools, i.e., no tool offers an IDE-integrated, powerful interactive visualization to the diagnostic reports produced by automatic fault localization techniques (Riboira, 2011). GZOLTAR offers an extendable visualization framework and is described using, as proof-of-concept, two potentially useful visualizations for the diagnostic reports.

GZOLTAR provides a quick view of a project structure, the relationships between lines of code, and the probability of each component to be at fault. GZOLTAR is integrated in Eclipse, a popular IDE (Geer, 2005). GZOLTAR uses Eclipse's standard features, such as integration with code editors and standard Eclipse warnings generation, and offers interactive visualizations of the system under test, directly inside an Eclipse view. Users can swap between visualizations to better understand the system architecture and failure distribution among it. Users can also navigate through the visualizations to analyze in detail a specific area of the system under test. They can use some navigation techniques such as node expand/ collapse, zoom and pan, root change, and can see relations between lines of code. They can also jump directly to the software source code to fix quickly the identified fault.

The effectiveness of the presented tool was assessed with a usability test, performed with

a small group of informatics engineering MSc and PhD students. The results of this study were very positive, and shown that users found GZOLTAR to be a powerful debugging tool. The study was also useful to get users' feedback to aid in future developments of this tool.

Being an active project, GZOLTAR can be enhanced in many ways. First, we are taking into account the feedback of the user study to improve the plug-in. Second, we plan to add new visualizations to the framework, and provide some features which are useful to all visualizations, like a mini-map to be displayed when the user zooms into a specific system area (showing the entire system and the area that has been zoomed in), and a color spectrum bar to aid the user to better identify the failure probability of each component. Finally, further user testing and enquiries are also required, to evaluate in a meaningful way the effectiveness of the new additions to this system, and to collect more ideas for different types of visualization that may be identified as useful.

### REFERENCES

Abreu, R. (2009). *Spectrum-based fault localization in embedded software*. PhD Thesis, Delft University of Technology, NL.

Abreu, R., Zoeteweij, P., & Van Gemund, A. J. C. (2009). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, *82*(11), 1780–1792. doi:10.1016/j.jss.2009.06.035.

Agrawal, H., de Millo, R., & Spafford, E. (1991). An execution backtracking approach to program debugging. *IEEE Software*. doi:10.1109/52.88940.

Balzer, R. M. (1969). EXDAMS: Extendible debugging and monitoring system. In *Proceedings of the AFIPS Spring Joint Conference, AFIPS Press*.

Bouillon, P., Krinke, J., Meyer, N., & Steimann, F. (2007), Ezunit: A framework for associating failed unit tests with potential programming errors. In *Proceedings of the International Conference on Agile Processes in Software Engineering and Extreme Programming (XP'07)*. Springer.

Burnette, E. (2005). *Eclipse IDE pocket guide*. O'Reilly Media, Inc..

Dale, C., & Anderson, T. (2009). In Proceedings of the Seventeenth Safety-Critical Systems Symposium on Safety-Critical Systems: Problems, Process and Practice. Springer Publishing Company, Inc.

Geer, D. (2005). Eclipse becomes the dominant Java IDE. *Computer*, *38*(7), 16–18. doi:10.1109/MC.2005.228.

Hailpern, B., & Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, *41*(1), 4–12. doi:10.1147/sj.411.0004.

Harrold, M., Rothermel, G., Wu, R., & Yi, L. (1998). An empirical investigation of program spectra. ACM SIGPLAN Notices.

Hoffmann, M. R. (2011), *JaCoCo*. Retrieved from http://www.eclemma.org/jacoco/

Janssen, T., Abreu, R., & Van Gemund, A. J. C. (2009), Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (pp. 662–664). Washington, DC: IEEE Computer Society.

Johnson, B., & Shneiderman, B. (1991). Tree-maps: A spacefilling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd Conference on Visualization*, Los Alamitos, CA (pp. 284–291). IEEE Computer Society Press.

Jones, J. A., Harrold, M. J., & Stasko, J. T. (2002). Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE'02)*, ACM Press.

Ko, A. J., & Myers, B. A. (2009). Finding causes of program output with the Java Whyline. In *Proceedings of the Human Factors in Computing Systems (CHI'2009)* (pp. 1569-1578).

Lintern, R., Michaud, J., Storey, M.-A., & Wu, X. (2003). Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM Symposium on Software Visualization (SoftVis'03)* (pp. 47–56). New York, NY: ACM.

McCullough, M. (2006). Developing eclipse plugins. *Linux Journal*, 143, 11.

Nielsen, J., & Landauer, T. K. (1993). A mathematical model of the finding of usability problems. *Transport*, 206, 206–213.

O'Neal, M., & Stewart, T. (1996). *Awt programming for Java* (1st ed.). New York, NY: Henry Holt and Co. Inc..

Copyright © 2012, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., & Cutler, J. ... Treuhaft, N. (2002), Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies, *Technical Report UCB/CSD-02-1175*, University of California at Berkeley, CA.

Riboira, A. (2011). *GZoltar: A graphical debugger interface*. Master's thesis, University of Porto, Portugal.

Riboira, A. (2011). *GZoltar: Fixing faults (vid-eo)*. Retrieved from http://www.youtube.com/ watch?v=JkimgY0NGSc

RTI. (2002). Planning report 02-3: The economic impacts of inadequate infrastructure for software testing. Planning report, National Institute of Standards and Technology.

Shreiner, D., & Group, T. K. O. A. W. (2009). *OpenGL* programming guide: The official guide to learning *OpenGL*, Versions 3.0 and 3.1 (7th ed.). Addison-Wesley Professional.

Stallman, R. (1994). *Debugging with GDB - The GNU* source level debugger. Free Software Foundation.

Stasko, J., Catrambone, R., Guzdial, M., & Mc-Donald, K. (2000). An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies*, 53(5), 663–694. doi:10.1006/ ijhc.2000.0420. Sun Microsystems, Inc. (1990). DBX. - Debugging tools DBX, SunOS 4.1.1 ed.

Van Wijk, J. (2005). The value of visualization. In *Proceedings of the Visualization 2005 (VIS'05)* (pp. 79–86).

Wolff, D. (2005). Using opengl in java with jogl. *The Journal of Computing Science in Small Colleges*, 21, 223–224.

Zeller, A., & Lütkehaus, D. (1996). *DDD - A free graphical front-end for UNIX debuggers*. ACM SIGPLAN Notices. doi:10.1145/249094.249108.

## **ENDNOTES**

- http://www.eclipse.org/ (May, 2012)
- <sup>2</sup> http://www.microsoft.com/visualstudio/ (May, 2012)
- http://developer.apple.com/xcode/ (May, 2012)
- <sup>4</sup> http://www.embarcadero.com/products/delphi/ (May, 2012)
  - http://devkix.com/nanoxml.php (May, 2012)

André Riboira graduated in Informatics Engineering (BSc) at the Higher Institute of Engineering of Porto, Portugal, in 2006, and in Informatics and Computing Engineering (MSc) at the Faculty of Engineering of the University of Porto, Portugal, in 2011, with a thesis on automatic debugging. He worked as a software developer for the Faculty of Medicine of the University of Porto, and also as a freelancer software developer for 6 years. He founded its own company in 2007 focused in web application development, where he was manager for 4 years. During 2011 he was a researcher at the University of Minho and enrolled on doctoral program on Informatics Engineering at the Faculty of Engineering of the University of Porto, where he is currently a researcher in automatic testing and debugging.

Rui Rodrigues graduated in Systems and Informatics Engineering at Minho University in 1998. During his PhD he researched in the area of 3D reconstruction from Images divided between Philips Research, Eindhoven, and Minho University, until he concluded in 2006. He worked in the industry in the field of interactive systems, until he joined FEUP as Invited Assistant Professor in 2009, to teach and research in the areas of Computer Graphics, Interaction and Gaming.

Copyright © 2012, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.

Rui Abreu graduated in Systems and Informatics Engineering at University of Minho, Portugal, in 2004, carrying out his graduation thesis project at Siemens S.A., Portugal. Between September 2002 and February 2003, Rui followed courses of the Software Technology Master Course at Utrecht University, the Netherlands, as an Erasmus Exchange Student. He was a (student) researcher at Philips Research Labs, the Netherlands, between October 2004 and June 2005. In 2009, he concluded his PhD in Computer Science at the Deflt University of Technology, the Netherlands. He is currently with the Faculty of Engineering of University of Porto as an Assistant Professor in Software Engineering.

José Campos concluded his MSc in Informatics and Computing Engineering at Faculty of Engineering of University of Porto, Portugal in 2012. During his MSc he investigated in the area of regression testing that focusing primarily on the reduction of test suites, with the main goal of reducing the cost of re-testing a software program. Currently he is a researcher at the Faculty of Engineering of University of Porto in a project related with automatic error detection in software.