Interactive visualizations of automatic debugging reports

André Riboira[1], Rui Rodrigues[1] & Rui Abreu[1]

[1]Dept. of Informatics Engineering, Faculty of Engineering, University of 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, current automatic debugging tools suffer from a common shortcoming that may affect their adoption and correct use: the lack of effective visualizations of their output. This paper explores the use of interactive visualization techniques applied to automatic debugging results, and integrated in a common development environment (Eclipse), to improve the efficiency of the debugging process. The proposed tool – GZOLTAR– *is an Eclipse plug-in that includes an automatic debugging algorithm and generates interactive visualizations of the resulting hierarchical data, with direct links to the IDE's source code editor.*

1. Introduction

Software faults are among the strongest concerns of software developers. Unfortunately, almost all software projects have faults, and their localization and fixing is one of the most expensive tasks during software development [HS02]. On large systems, software faults are even more difficult to find, because they are usually not isolated, and their origin can be related with different system components. There are also systems where a software fault can be critical, such as in the aviation context, or in military advanced equipment [DA09]. Due to this, methods and techniques that aid in fault localization and fixing are of utmost importance. Debugging techniques have therefore been the focus of many researchers. As an example, automated debugging techniques based on statistical analysis of historical test executions data have recently received considerable attention due to their diagnostic capabilities [Abr09]. However, current debugging tools (that implement these techniques) suffer from a common shortcoming that may affect their adoption and correct use, which is the lack of effective visualizations of their output [Rib11a].

Visualization is very important in information comprehension [vW05]. In general, human beings find it more intuitive to understand information laid out in a logical, hierarchical way, than with a simple a list of values. The tool presented in this paper – GZOLTAR– was created to fill this gap, by offering a package with automatic debugging and interactive debugging data visualizations to be seamlessly added to a developer's integrated development environment (IDE). The main premises in GZOLTAR's development were therefore the following:

- To implement a robust automatic debugging framework that allows different visualization techniques, and that may be easily expanded in the future;
- To help the user to find software faults faster, by aiding the understanding of debugging results;
- To 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;
- To have an easy and fast installation process to facilitate its adoption and use;

The automatic debugging tool that is behind GZOLTAR is called ZOLTAR [JAG09], a Spectrum-Based Fault Localization (SFL) framework whose performance is among the best [AZGvG09]. The chosen IDE was Eclipse [Bur05] due to its wide adoption [Gee05] and its plug-in development facilities [McC06]. The interactive visualization framework uses OpenGL for graphics rendering, due to its flexibility to produce both 2D and 3D graphics, its performance supported by hardware acceleration and its multi-platform availability [SG09]. As Eclipse can not access OpenGL directly, some supporting libraries such as JOGL were used to create bindings to OpenGL native system libraries [Wol05]. The interactive visualization framework is extendable, and on this paper we present two examples of possible visualizations.

GZOLTAR processing can be divided into three phases:

- **Project and Code Detection** on Eclipse;
- **Automatic Debugging Process** using the collected data;
- **Interactive Visualization Framework** in Eclipse view.

The Interactive Visualization Framework creates different debugging data visualizations and allows navigation and integration with default Eclipse features, such as the code editor and the building warnings' list.

2. Automatic Debugging using the Zoltar Framework

GZOLTAR's automatic debugging core, ZOLTAR, is an actively-developed framework that performs statistical analysis of historical software test executions' data to calculate the failure probability of each component of a system under test (SUT). That calculation is made using spectrum-based fault localization (SFL) algorithms [JAG09]. ZOLTAR's SFL algorithms are among the most efficient [AZGvG09], and its usefulness has already been demonstrated and recognized on academic and industrial environments, and is currently under active development.

2.1. Concepts

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 gets recorded. This record is called 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 coverage 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) [Abr09].

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

Figure 1: ***SFL input matrix.*** *N means test executions, M means SUT components, a means code coverage and e means test execution result.*

Because ZOLTAR is based on test executions code coverage and results, its accuracy will depend on the quality of test cases. The ZOLTAR framework offers several SFL algorithms, including the Ochiai algorithm [AZGvG09] (amongst the best for diagnosis). 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). Besides, at this moment XZOLTAR is only available for Linux operating system. 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.

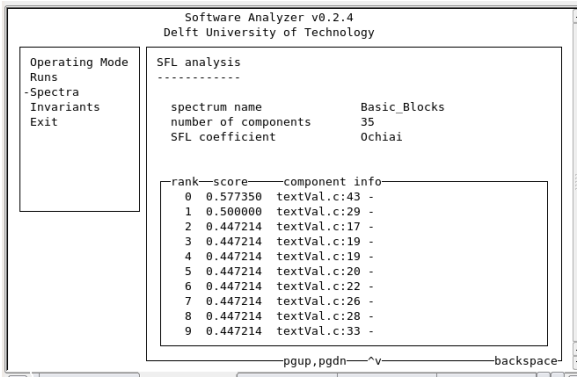

Figure 2: *Zoltar output.*

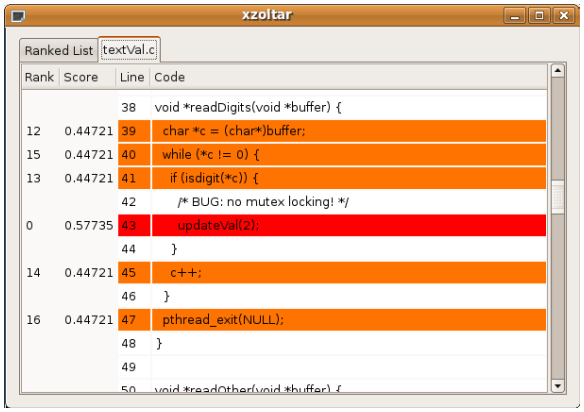

Figure 3: *XZoltar output.*

3. Visualizations of debugging results

As stated earlier, current automatic debugging tools (including Zoltar) lack efficient visualizations and integration with IDE's. GZOLTAR's purpose is to address this issue, by laying out the debugging information in context with the SUT code structure. Under Eclipse and in particular in terms of Java development, the paradigm of workspace is used. 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 ZOLTAR'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 [SCGM00]. A given representation may be effective for a tree with a given pattern of node weight distribution and node relationships, but it would be too confuse for a differently-structured tree. Furthermore, different people react differently to the same visualization, so it is important to give alternatives to the user [SCGM00].

For this reason, GZOLTAR was implemented in a modular way that allows to easily add different visualizations to the system. In the current version, two visualizations were implemented, treemap and sunburst. 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.

3.1. Sunburst

Sunburst is a circular visualization [SCGM00] 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, ...) have their area calculated based on the sum of the areas of their descendants (see Figure 4). 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 each line of code of the system.

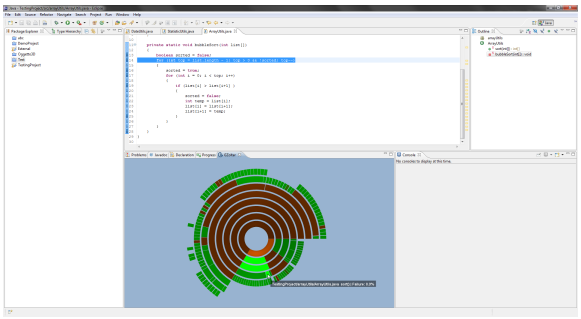

Figure 4: GZOLTAR *sunburst visualization.*

The color of each leaf node will vary from the one that the selected component has to gray, revealing the percentage of test executions where that line and the selected one were both executed. 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.

3.2. Treemap

Treemap is a rectangular visualization [JS91] 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 at odd levels of the hierarchy are divided horizontally and nodes at even levels are divided vertically (see Figure 5). With this concept the leaves have the majority of the display area (the remaining corresponds to the margins).

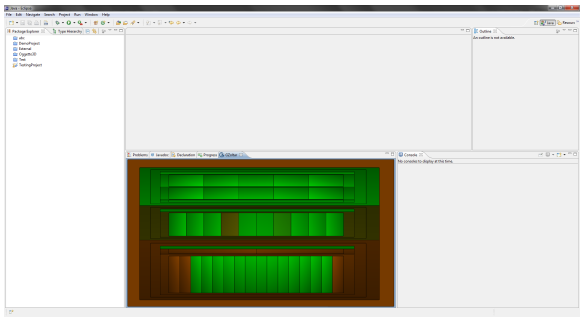

Figure 5: GZOLTAR *treemap visualization.*

3.3. Comparison

These two visualization concepts privilege 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, ...) 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. 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 6. 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'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 7, 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.

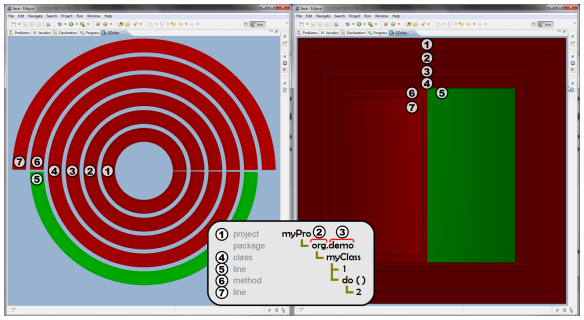

Figure 6: GZOLTAR *visualizations comparative.*

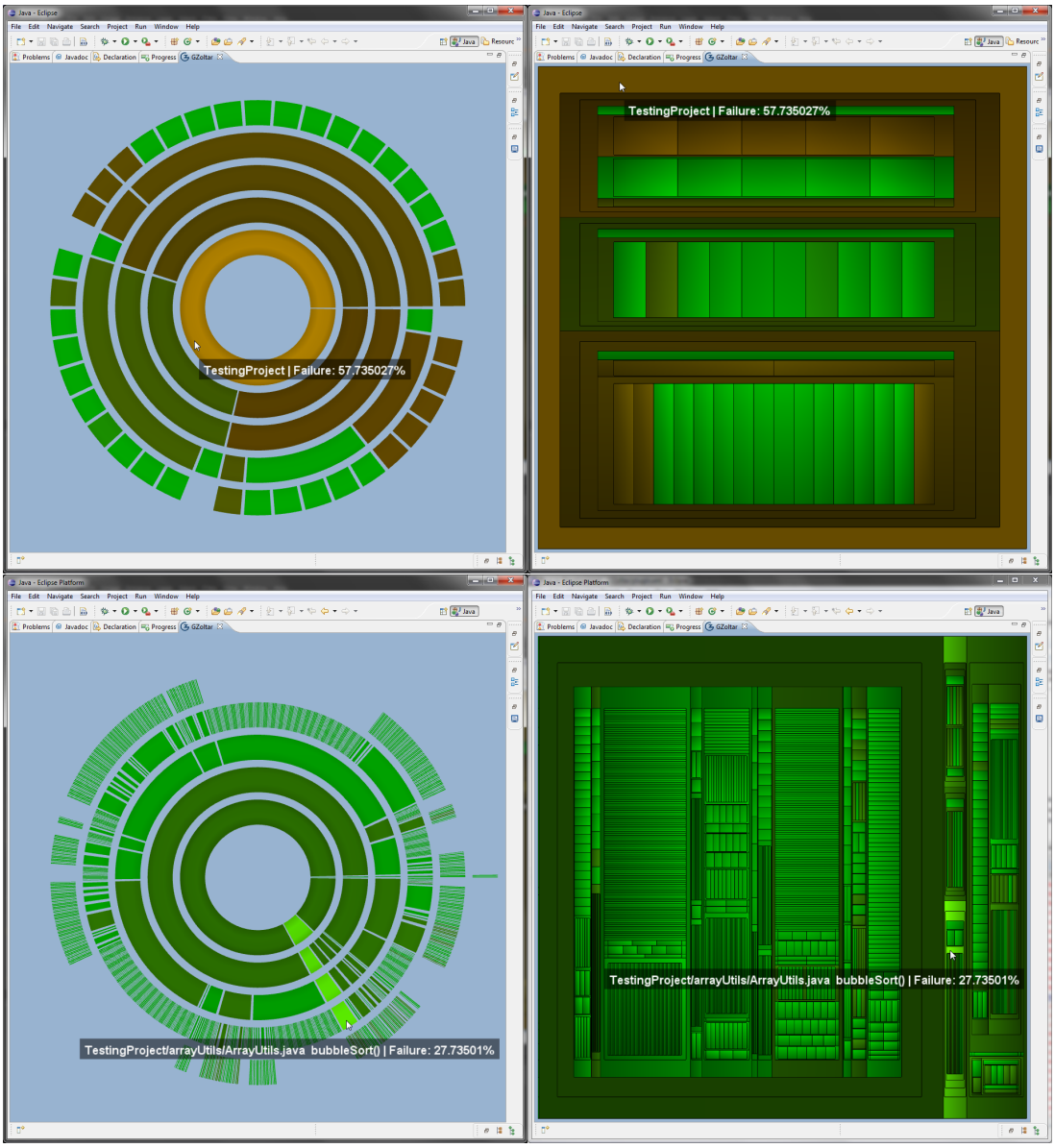

Figure 7: GZOLTAR *visualizations of a simple (top) and complex (bottom) systems.*

4. 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 highlight, but also about project organization, code completion, integrated help and the ability to analyze the system state at a given stage. The most powerful automatic debugging tools are external to IDE's, which may compromise their adoption. GZOLTAR is integrated with 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 his/her 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 8). For interested readers, refer to [Rib11b] for a video demonstration GZOLTAR.

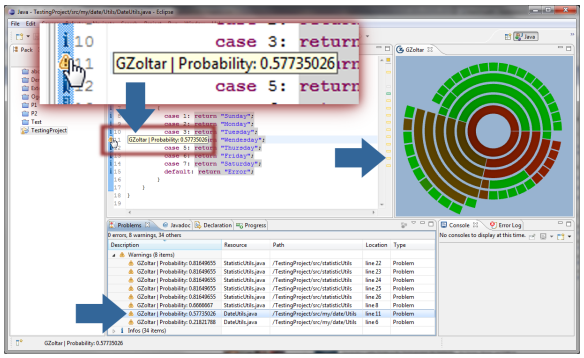

Figure 8: GZOLTAR *integration with Eclipse IDE.*

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.

4.1. 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 9). If the user clicks on a node that is already open, he will collapse it. Pressing the “space” key will expand all nodes. If the user presses the “space” key again he 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 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.

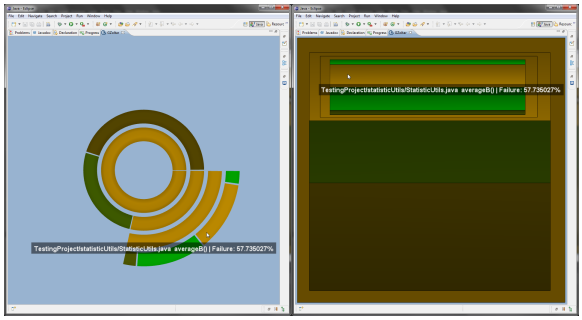

Figure 9: GZOLTAR *navigation.*

4.2. Zoom and pan

One useful feature that can help 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 double clicking on a visualization spot without releasing the mouse button on second click. Panning is also possible by clicking on a visualization spot and drag the visualization without releasing the left mouse button. By zooming in and pan to the desired place, the user can click easily on tiny nodes, or analyze in detail a small portion of the system (see Figure 10).

4.3. Root Change

Another feature useful for large projects is root change. The user can chose any inner tree node 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 all 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 11). 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.

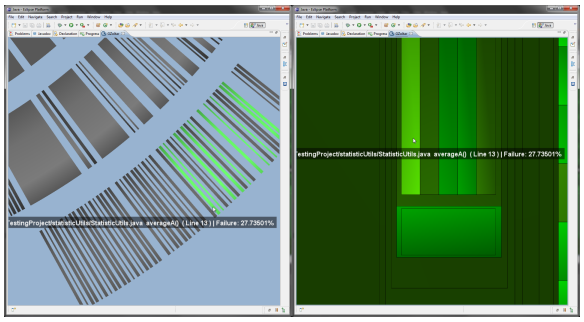

Figure 10: GZOLTAR *zoom.*

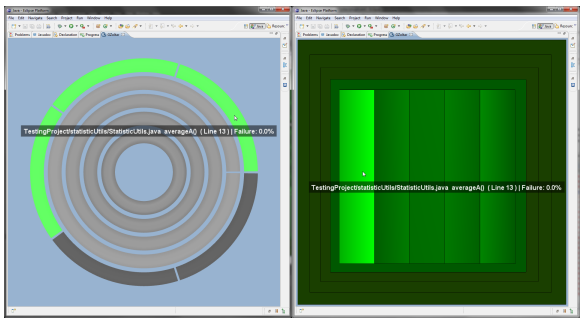

Figure 11: GZOLTAR *root change.*

5. GZOLTAR Architecture

GZOLTAR is an Eclipse Plugin [McC06] 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 user interface. Because GZOLTAR’s output is an Eclipse view, it uses the toolkit that produces Eclipse views SWT which is part of Eclipse’s Workbench [LMSW03]. 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. Although OpenGL is not available directly for Java, there is a tool, JOGL [Wol05], that provides OpenGL library bindings to Java. To render OpenGL scenes, JOGL uses Java’s AWT [OS96], which is a multiplatform toolkit to produce Java GUI’s [LMSW03]. 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 [Hof11]. For a representation of GZOLTAR technological layers see Figure 12.

6. Conclusions and Future Work

GZOLTAR goal was to fill a gap in current automatic debugging tools, i.e., no tool offers a powerful visualization to the diagnostic reports. In this paper, we presented a tool, coined GZOLTAR, that offers an extendable visualization framework with two examples of possible visualizations.

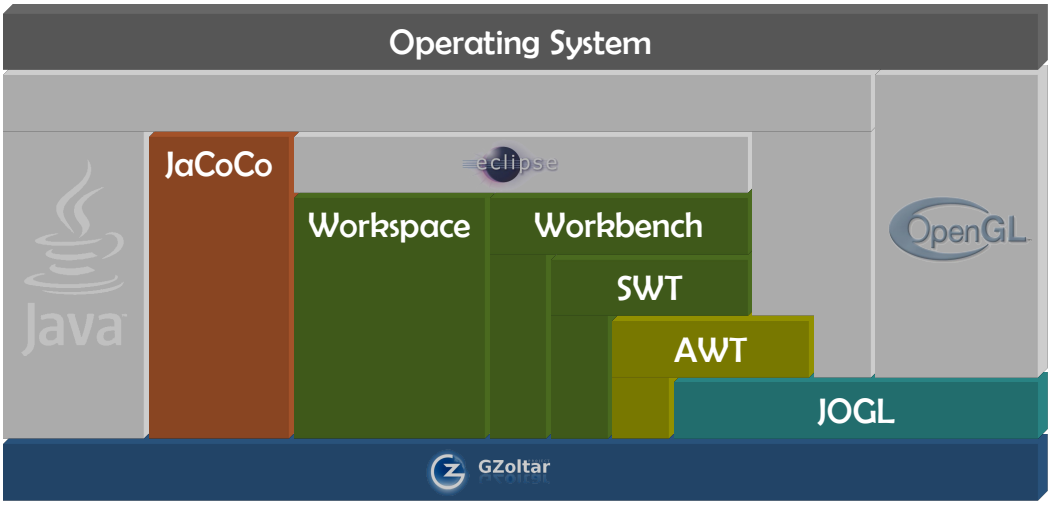

Figure 12: GZOLTAR *technological layers.*

GZOLTAR provides a quick view of a project structure, the relations of its lines of code, and the probability of each component to fail. GZOLTAR is integrated in Eclipse, a popular IDE [Gee05]. GZOLTAR uses Eclipse default 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 failures distribution among it. Users can also navigate through the visualizations to analyze in detail a specific area of the system under test.

GZOLTAR is an active project, it can be enhanced in many ways. New visualizations will be added to GZOLTAR framework. Some features common to all visualizations will also be developed, 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. New interaction concepts will also be explored, such as multi-touch devices. The Ochiai algorithm can be parallelizable, so, some GPGPU techniques [WL08] can also be explored to optimize its processing. Testing with humans will also be done, to analyze the efficiency of the different visualizations on fault localization. These tests will also help to define future GZOLTAR developments.

References

[Abr09] ABREU R.: *Spectrum-based Fault Localization in Embedded software*. PhD in computer science, Delft University of Technology, 2009. 1, 2

[AZGvG09] ABREU R., ZOETEWEIJ P., GOLSTEIJN R., VAN GEMUND A. J. C.: A practical evaluation of spectrum-based fault localization. *J. Syst. Softw. 82*, 11 (2009), 1780–1792. 1, 2

[Bur05] BURNETTE E.: *Eclipse IDE Pocket Guide*. O'Reilly Media, Inc., 2005. 1

[DA09] DALE C., ANDERSON T.: *Proceedings of the Seventeenth Safety-Critical Systems Symposium on Safety-Critical Systems: Problems, Process and Practice*. Springer Publishing Company, Incorporated, 2009. 1

[Gee05] GEER D.: Eclipse becomes the dominant Java IDE. *Computer 38*, 7 (2005), 16–18. 1, 6

[Hof11] HOFFMANN M. R.: Jacoco. http://www.eclemma.org/jacoco/, Jan 2011. 5

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

[JAG09] JANSSEN T., ABREU R., GEMUND A. J. C. V.: Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering* (Washington, DC, USA, 2009), IEEE Computer Society, pp. 662–664. 1, 2

[JS91] JOHNSON B., SHNEIDERMAN B.: Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2nd Conference on Visualization* (Los Alamitos, CA, USA, 1991), IEEE Computer Society Press, pp. 284–291. 3

[LMSW03] LINTERN R., MICHAUD J., STOREY M.-A., WU X.: Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proceedings of the 2003 ACM symposium on Software visualization* (New York, NY, USA, 2003), SoftVis '03, ACM, pp. 47–ff. 5

[McC06] MCCULLOUGH M.: Developing eclipse plugins. *Linux J. 2006*, 143 (2006), 11. 1, 5

[OS96] O'NEAL M., STEWART T.: *Awt Programming for Java*, 1st ed. Henry Holt and Co., Inc., New York, NY, USA, 1996. 5

[Rib11a] RIBOIRA A.: *GZoltar: A Graphical Debugger Interface*. Master's thesis, University of Porto, 2011. 1

[Rib11b] RIBOIRA A.: Gzoltar: Fixing faults (video). http://www.youtube.com/watch?v=JkimgY0NGSc, Feb 2011. 4

[SCGM00] STASKO J., CATRAMBONE R., GUZDIAL M., MCDONALD K.: An evaluation of space-filling information visualizations for depicting hierarchical structures. *International Journal of Human-Computer Studies 53*, 5 (2000), 663–694. 2, 3

[SG09] SHREINER D., GROUP T. K. O. A. W.: *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*, 7th ed. Addison-Wesley Professional, 2009. 1

[vW05] VAN WIJK J.: The value of visualization. In *Visualization, 2005 (VIS'05)* (2005), IEEE Computer Society, pp. 79 – 86. 1

[WL08] WU E., LIU Y.: Emerging technology about gpgpu. In *Proceedings of the IEEE Asia-Pacific Conference on Circuits and Systems* (2008), pp. 618–622. 6

[Wol05] WOLFF D.: Using opengl in java with jogl. *J. Comput. Small Coll. 21* (October 2005), 223–224. 1, 5