Using HTML5 Visualizations in Software Fault Localization

Carlos Gouveia, José Campos, Rui Abreu
Department of Informatics Engineering
Faculty of Engineering, University of Porto
Porto, Portugal
{carlos.gouveia, jose.carlos.campos}@fe.up.pt, rui@computer.org

***Abstract*—Testing and debugging is the most expensive, error-prone phase in the software development life cycle. Automated software fault localization can drastically improve the efficiency of this phase, thus improving the overall quality of the software. Amongst the most well-known techniques, due to its efficiency and effectiveness, is spectrum-based fault localization. In this paper, we propose three dynamic graphical forms using HTML5 to display the diagnostic reports yielded by spectrum-based fault localization. The visualizations proposed, namely Sunburst, Vertical Partition, and Bubble Hierarchy, have been implemented within the GZOLTAR toolset, replacing previous and less-intuitive OpenGL-based visualizations. The GZOLTAR toolset is a plug-and-play plugin for the Eclipse IDE to ease world-wide adoption. Finally, we performed an user study with GZOLTAR and confirmed that the visualizations help to drastically reduce the time needed in debugging (e.g., all participants using the visualizations were able to pinpoint the fault, whereas of those using traditional methods only 35% found the fault). The group that used the visualizations took on average 9 minutes and 17 seconds less than the group that did not use them.**

***Index Terms*—Automatic Debugging; Reports; Visualizations; GZOLTAR.**

I. INTRODUCTION

It is a fact of life that most software systems (except for just a few basic systems in controlled environments) have a high probability of containing faults, being many of them detected only after a long period of use. Moreover, the testing and debugging phase is the most unpredictable with respect to effort and costs of the software development life-cycle [1]. Despite the number of automatic testing and debugging techniques proposed in recent years, software developers still perform it mainly using manual approaches, such as prints in the code and breakpoints. The lack of integration between the multiple automatic testing and debugging techniques hinders world-wide adoption by the developers.

Recently, we developed a toolset aiming at helping the developer through this, rather cumbersome, testing and debugging phase. The tool - coined GZOLTAR[1] [2] - is offered as a plugin for the well-known Eclipse Integrated Development Environment (IDE) [3]. It runs on Microsoft Windows, Apple Mac OS and Linux, either 32 or 64-bit architecture systems.

[1]Note that GZOLTAR also provides techniques for managing regression testing of JUnit test suites (such as minimization), which can potentially reduce the effort (in terms of time) for re-testing the software system.

GZOLTAR takes as input the coverage information of the executed test cases to run a Spectrum-based Fault Localization (SFL) technique [4]. The SFL technique offered by GZOLTAR is the Ochiai, first proposed in [5], and known to be amongst the best SFL techniques available. This technique yields a ranked list in order of suspiciousness of a component (a component may be a project, a package, a class, a method or a line) being faulty. Moreover, the GZOLTAR toolset resorted to Open Graphics Library (OpenGL) to render the diagnostic report's visualizations. In particular, it offered the user to render the diagnostic reports using either Sunburst or Treemap. The Sunburst was re-implemented and the Treemap was abandoned. However, our OpenGL implementation was not optimal, with several shortcomings. When more complex visualizations were rendered, the scene became heavy in terms of computational resources used, entailing a negative impact in the user interaction. Besides, there are a few incompatibilities with some Graphical Processing Unit (GPU)'s and graphic cards drivers originating a defective rendered scene.

To address these issues, the OpenGL-based visualizations were replaced with visualizations in Hypertext Markup Language, version 5 (HTML5): (1) to reduce the computational resources needed, (2) to improve the interaction with the user with new options, and (3) to even offer more visualizations. In particular, we considered the D3.js [6] JavaScript library to display digital data in dynamic graphical forms.

The HTML5 visualizations provided by GZOLTAR, a novel concept in the context of fault localization, aid developers to pinpoint the most suspicious system components. Furthermore, users are able to interact with the visualizations in order to isolate the faulty component. The IDE's editor also displays warnings next to the lines that are considered faulty by the fault localization technique for the developer to quickly spot suspicious parts. The toolset creates an ideal and integrated ecosystem to manage tasks related to testing and debugging tasks (from the execution of JUnit tests to the source code analysis) in order to help identifying and fixing the faults responsible for observed failures.

We performed a user study with GZOLTAR and confirmed that the features provided by the toolset are of great benefit when testing and debugging a software program. Building on top of Parnin and Orso's work [7], who concluded that only inspecting the rankings yielded by current fault localization

techniques may not always be sufficient, we agreed that the visualization of the diagnostic report adds extra information leading the developer to the fault quickly.

To sum up, the main contributions of this paper are:

- We propose a set of visualizations using HTML5 to display information-rich diagnostic reports. In particular, we implemented and studied the added value of the following visualizations: Sunburst, Vertical Partition, and Bubble Hierarchy;
- We incorporate the visualizations in a toolset coined GZOLTAR, which is publicly available;
- We carried out an user study, showing the benefits the visualizations bring to the debugging phase. Of the participants which did not use visualizations to aid the fault localization, only 35% found the fault, and even those had higher fault localization times than those using GZOLTAR. The group that used the visualizations took on average 9 minutes and 17 seconds less than the group that did not use them.

The reminder of this paper is organized as follows. Section II presents the technique of fault localization used in the GZOLTAR toolset. Section III describes the visualizations available in GZOLTAR and Section IV presents the functionalities of the toolset. Section V reports the findings of the user study carried out to validate the efficiency and effectiveness of the use of visualizations to support software fault localization and the usability of the GZOLTAR toolset, presenting the results obtained and discussing the feedback given by the users. Section VI discusses related work. Finally, Section VII concludes this paper.

II. FAULT LOCALIZATION

(Semi-)automatic fault localization has been an active area of research in recent years [8], [9], [10], [11], [12], [13]. This paper considers SFL as underlying, fault localization technique, which is amongst the best fault localization techniques [4]. Throughout this paper, we use the following terminology [14]:

- A *failure* is an event that occurs when delivered service deviates from correct service.
- An *error* is a system state that may cause a failure.
- A *fault* (defect/bug) is the cause of an error in the system.

In this paper, we apply this terminology to software programs, where faults are bugs in the program code. Failures and errors are symptoms caused by faults in the program. The purpose of fault localization is to pinpoint the root cause of observed symptoms.

Definition 1 *A software program* Π *is formed by a sequence* M *of one or more statements.*

Given its dynamic nature, central to the fault localization technique considered in this paper is the existence of a test suite.

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

Figure 1: Input for SFL technique: a_{ij} represents the coverage for component j in run i. Error detection e_i is true if test fails or false otherwise.

Definition 2 *A test suite* $T = \{t_1, \ldots, t_N\}$ *is a collection of test cases that are intended to test whether the program follows the specified set of requirements. The cardinality of* T *is the number of test cases in the set* $|T| = N$.

Definition 3 *A test case* t *is a* (i, o) *tuple, where* i *is a collection of input settings or variables for determining whether a software system works as expected or not, and* o *is the expected output. If* $\Pi(i) = o$ *the test case passes, otherwise fails.*

A. Program Spectra

A program spectrum is a characterization of a program's execution on a dataset [15]. This collection of data, gathered at runtime, provides a view on the dynamic behaviour of a program. The data consists of counters or flags for each software component. Various different program spectra exist [16], such as path-hit spectra, statement-hit-spectra, data-dependence-hit spectra, and block-hit spectra. In the context of this paper we consider statement-hit-spectra.

In order to obtain information about which components were covered in each execution, the program's source code needs to be instrumented, similarly to code coverage tools [17]. This instrumentation will monitor each component and register those that were executed (see Figure 1). Components can be of several detail granularities, such as classes, methods, or statements.

B. Fault Localization

Spectrum-based Fault Localization (SFL) exploits information from passed and failed system runs. A passed run is a program execution that is completed correctly, and a failed run is an execution where an error was detected [4]. The criteria for determining if a run has passed or failed can be from a variety of different sources, namely test case results and program assertions, among others. The information gathered from these runs is their hit spectra [4].

The hit spectra of N runs constitutes a binary $N \times M$ matrix A, where M corresponds to the instrumented components of the program. Information of passed and failed runs is gathered in a N-length vector e, called the error vector. The pair (A, e) serves as input for the SFL technique, as seen in Figure 1.

With this input, fault localization consists in identifying what columns of the matrix A resemble the vector e the most. For that, several different similarity coefficients can be

	public Complex reciprocal()	Tests						
	{	1	2	3	4	5	6	s_O
1:	if (isNaN)	●	●	●	●	●	●	0.408
2:	return NaN;					●		0.000
3:	if (real == 0.0 && imaginary == 0.0)	●	●	●	●		●	0.447
4:	return NaN; /* FAULT */						●	1.000
5:	if (isInfinite)	●	●	●	●			0.000
6:	return ZERO;				●			0.000
7:	if (FastMath.abs(real) < FastMath.abs(imaginary)) {	●	●	●				0.000
8:	double q = real / imaginary;	●		●				0.000
9:	double scale = 1.0 / (real * q + imaginary);	●		●				0.000
10:	return createComplex(scale * q, -scale);	●		●				0.000
11:	} else {		●					0.000
12:	double q = imaginary / real;		●					0.000
13:	double scale = 1.0 / (imaginary * q + real);		●					0.000
14:	return createComplex(scale, -scale * q);}		●					0.000
	Error vector:	✓	✓	✓	✓	✓	✗	

Figure 2: Coverage of `reciprocal` function from Apache Commons Math project with Ochiai coefficient.

used [18]. One of the most effective is the Ochiai coefficient [19], used in the molecular biology domain:

$$s_O(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}} \quad (1)$$

in this case, $n_{pq}(j)$ is the number of runs in which the component j has been touched during execution ($p = 1$) or not touched during execution ($p = 0$), and where the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(j)$ counts the number of times component j has been involved in failed executions, whereas $n_{10}(j)$ counts the number of times component j has been involved in passed executions. Formally, $n_{pq}(j)$ is defined as

$$n_{pq}(j) = |\{i \mid a_{ij} = p \wedge e_i = q\}|. \quad (2)$$

SFL can be used with program spectra of several different granularities. However, it is most commonly used at the statement or basic block level, as coarser granularities may make it difficult for programmers to investigate if a given fault hypothesis generated by SFL was, in fact, faulty. Throughout this work, we use statement level as the component granularity for the fault localization diagnosis report.

C. Example

Figure 2 presents an example of the SFL technique, using the Ochiai coefficient for the `reciprocal` function from the Apache Commons Math[2] project. This function returns the multiplicative inverse of `this` element. To improve legibility, the coverage matrix and the error detection vector are transposed. The detail granularity for each component of the hit spectra in this particular example is the line of code. The program contains a fault in line 4: it should read `return INF;`.

Six test cases were executed, and their coverage information for each line of code are on the right-hand-side of Figure 2.

[2]Apache Commons Math project homepage http://commons.apache.org/proper/commons-math/, 2013.

Table I: Faulty statement ranking.

Ranking	s_O		Statement
1	1.000	4:	return NaN; /* FAULT */
2	0.447	3:	if (real == 0.0 && imaginary == 0.0)
3	0.408	1:	if (isNaN)
4	0.000	2:	return NaN;
5	0.000	5:	if (isInfinite)
6	0.000	6:	return ZERO;
7	0.000	7:	if (FastMath.abs(real) < FastMath.abs(imaginary)) {
8	0.000	8:	double q = real / imaginary;
9	0.000	9:	double scale = 1.0 / (real * q + imaginary);
10	0.000	10:	return createComplex(scale * q, -scale);
11	0.000	11:	} else {
12	0.000	12:	double q = imaginary / real;
13	0.000	13:	double scale = 1.0 / (imaginary * q + real);
14	0.000	14:	return createComplex(scale, -scale * q);}

At the bottom there is also the test case pass/fail info for each execution - which corresponds to the error detection vector e. According to this pass/fail status, the test number 6 fails, and all the other ones pass. Then, the similarity coefficient was calculated for each line using the Ochiai coefficient (Equation 1). These results represent the suspiciousness of a certain line containing a fault. The higher the coefficient, the more likely it is that a line contains a fault. Therefore, these similarity coefficients can be ranked to form an ordered list of the probable faulty lines. This ranking can be seen in Table I. Note that lines whose coefficient is zero can be stripped out of the ranking. This is because they are not executed when a test fails, so they cannot be the cause of the abnormal behaviour.

For the example, the first element in the ranking (which has also the highest coefficient) is statement number 4 - the faulty statement. As such, the developer only has to inspect this position in the ranking to find the faulty statement. Although contrived, this example serves well to demonstrate how SFL works.

A shortcoming of presenting the information in such format is that users do not traverse the ranking line by line and/or fully understand the meaning of the coefficients [7]. Also, important

information is not given to the user with the ranking (e.g., topology of the program). Our previous attempt to present a visual representation of the ranking was using OpenGL [2], and entailed the following drawbacks: (1) resource intensive, (2) difficult to enhance with new visualization features, and (3) not straightforward to create new visualizations.

III. Diagnostic Report's Visualizations

To make the ranking yielded by the underlying fault localization technique (as in Table I) more understandable by the developer as well as enhance it with extra - and important - information, we propose diagnostic report's visualizations. The visualizations have the capability to display the report in an information rich setting, and display the structure of the program as an hierarchical structured tree. This helps developers to understand the data organization and to access the position of a component in the program. The values of the similarity coefficients for each component are used to create a *smooth* colour gradient for the component, varying from red to green: red means that component is likely responsible for the observed faults; green means that component is not responsible; and yellow is the mid value between red and green. The colour of a non-leaf element is determined by the maximum suspiciousness of all of its children.

The proposed visualizations can be divided into: (1) adjacency diagram and (2) enclosure diagram. Sunburst (see Figure 3a) and Vertical Partition (see Figure 3b) are adjacency diagrams, Bubble Hierarchy (see Figure 3c) is an enclosure diagram. All visualizations are space-filling variants of node-link diagrams, because the link between father and children disappear and are drawn only the nodes as solid spaces. In these cases, each node is drawn taking into account the size of its children, in other words, the size of the node is proportional to the size of the sub-tree that is derived from it. This way to represent the hierarchic tree has the advantage of passing a better idea of the relative dimensions of each component, because this information could be difficult to represent on a node-link diagram. Another advantage that this kind of adjacency and enclosure diagrams have over node-link diagrams is that node-link diagrams could waste much space to represent a big amount of hierarchical data.

A. Sunburst

The Sunburst (see Figure 3a) visualization uses arcs as solid areas which represent the nodes. The radius of each one proportionally varies with the size of the respective sub-tree. The root element is drawn always at the centre of the visualization, and the children are expanded outward from it. This visualization uses polar coordinates to properly position each arc. It is a popular visualization frequently used, for instance, to display information of hard disk drives.

B. Vertical Partition

Vertical Partition (see Figure 3b) visualization uses rectangles instead of arcs to represent each node, and the children of each parent are drawn directly below it. As in Sunburst, the size of each solid node proportionally varies with the size of the respective sub-tree. The root is always drawn at the top of the visualization, and all children are drawn below from there.

C. Bubble Hierarchy

Unlike Sunburst and Vertical Partition, Bubble Hierarchy (see Figure 3c) applies the concept of containment instead of adjacency and uses circles to represent each solid node. Each child is drawn inside the area which represents the father, in other words, the area is recursively divided in subareas and so on. To better manage the visualization area, if any component has only one child, the father is not drawn, being drawn only the child. It is a different idea from the other two visualizations, but the final purpose of the hierarchic structure representation is the same.

IV. Toolset

GZoltar [2] is an Eclipse plugin [3] for automatic testing and debugging. Currently, the framework is provided as a plugin and library, and integrates seamlessly with the JUnit framework. It is an evolution from Zoltar [8] and is implemented using Java language and analyses programs written in Java. To install the GZoltar toolset, users need to request a license at

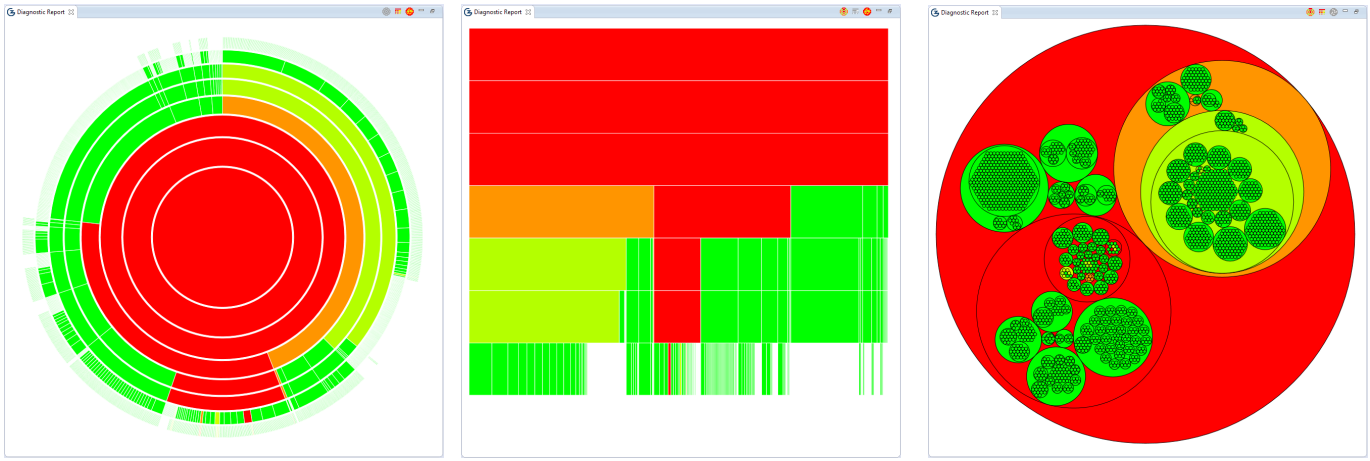
`http://www.gzoltar.com/`

Once the license is received, installing the software is straightforward: users only need to access to "Install New Software" sub-menu item (under the "Help" menu) and use the URL provided.

As said before, currently GZoltar offers three distinct visualizations implemented in HTML5: Sunburst (see Figure 3a), Vertical Partition (see Figure 3b) and Bubble Hierarchy (see Figure 3c). To select the desired visualization, the user can click at one of the three corresponding buttons at top right corner of the diagnostic report view. The order of the buttons is Sunburst, Vertical Partition and Bubble Hierarchy, from left to right. Note that this toolset uses a visualization framework, D3.js [6], which allows the creation of new visualizations with little effort. D3.js is a JavaScript (JS) library which allows the creation of different visual representations of data.

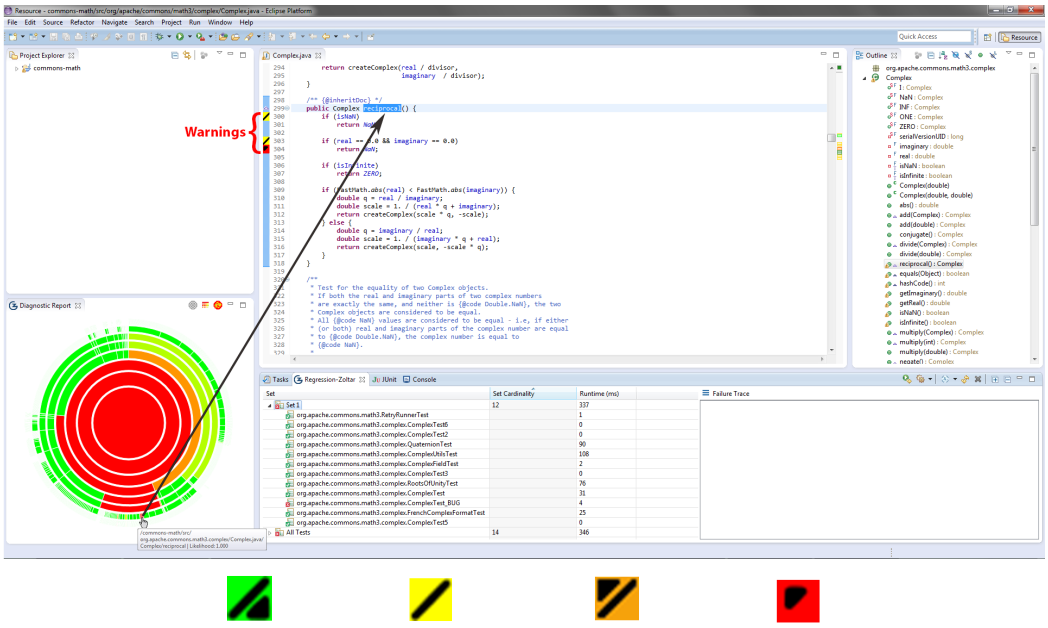
The generated visualizations are interactive, and the user is able to navigate through the project structure to analyse it in detail. The intention of the visualizations have the main goal of representing the analysed project in an hierarchical way (see Figure 5 for a graphical explanation) to allow a faster and easier debugging process. For instance, in the Sunburst visualization, each ring denotes an hierarchical level of the source code organization (from the inner to the outer circle).

All visualizations obey to a colour gradient ranging from green (low suspiciousness) to red (very high suspiciousness). The suspiciousness is computed by the diagnostic algorithm detailed in Section II-B. If all the tests passed (i.e., there were no observed failures), the underlying fault localization technique yields an empty diagnostic report. As such, GZoltar will render a visualization with all the system components in



(a) Sunburst (b) Vertical Partition (c) Bubble Hierarchy

Figure 3: GZOLTAR visualizations representing the structure of the Apache Commons Math project.



(a) Low Suspiciousness (b) Medium Suspiciousness (c) High Suspiciousness (d) Very High Suspiciousness

Figure 4: GZOLTAR global view representing the integration of toolset with the Eclipse IDE and warning description.

green. It should be noted that this does not mean that the system under test is bug-free, but rather that no failure was observed.

Users are able to navigate and interact with the visualizations. They may analyse each component by hovering the mouse cursor at any element on any visualization and a tooltip is shown with the identification of it and the corresponding suspiciousness value (see Figure 4). The user may click (`left-click`) on any component representation, and the code editor is automatically opened with the respective source code, as depicted in Figure 4. Users may also zoom in/out (with the mouse wheel) the visualization to analyse in detail a specific part of the system, and pan the visualization (see Figure 6 for more detail). The user may also resize the diagnostic report view at any time to automatically enlarge or reduce the visualization used.

Users are able to select any inner component to be the new root of the visualization, and only the sub-tree related to that component is displayed. This step was called “Root Change”, and can be seen as a “smart zoom”, because the viewing area gets limited to increase visualization detail, maintaining the same visual structure concept. To activate this feature the user hovers the mouse cursor at any element on any visualization, and double click with `left-click`. Thus all

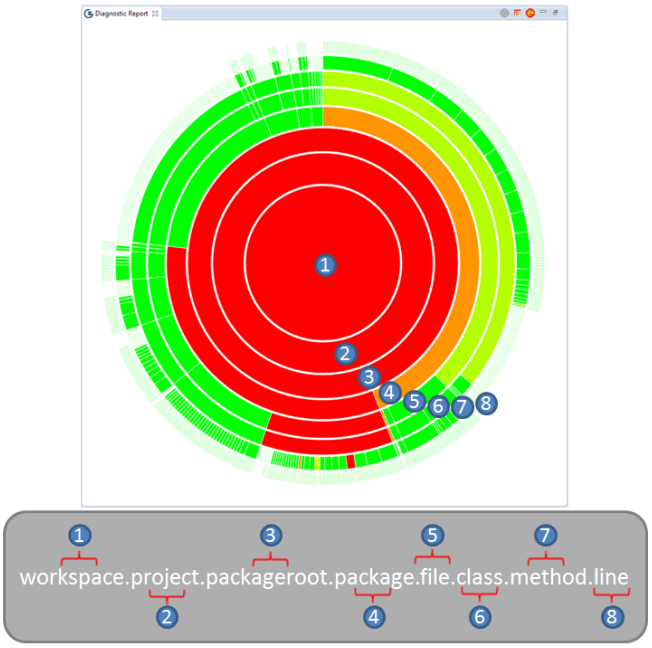
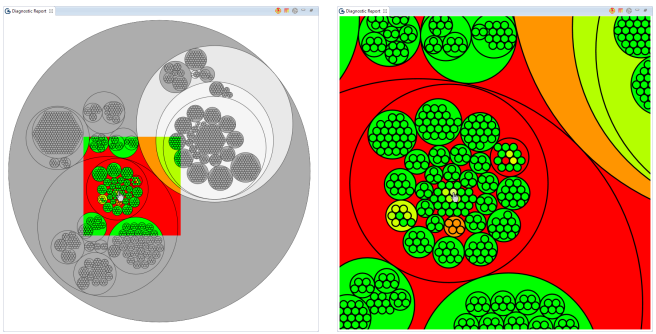


Figure 5: GZOLTAR hierarchical representation.



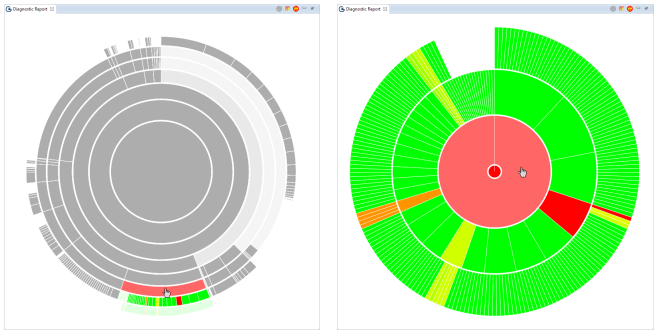
(a) Initial (b) Zoomed In

Figure 6: Zoom Feature.

the elements that are not related directly with the descendants or ascendants of the selected element will be hidden from the visualization (see Figure 7). This feature is available in all visualizations. `Right-click` reverts to the initial state, or double `left-click` on any higher hierarchical component to centre the zoom on it.

The GZOLTAR toolset also places warnings on the vertical ruler of the code editor next to the lines that are most likely to contain the fault. This list of warnings aid the developer in the process of pinpointing the faulty statement. The warnings can be of four types (see Figure 4): (1) red (Figure 4d) for the top lines most likely to contain a fault, (2) orange (Figure 4c) for high suspiciousness, (3) yellow (Figure 4b) for medium suspiciousness, and (4) green (Figure 4a) for low suspiciousness. Each warning embeds a ColorADD symbol[3], aimed at aiding colour-blind people distinguish between the

[3]ColorADD colour identification system, http://coloradd.net, 2013.



(a) Initial (b) Root Change

Figure 7: Root Change Feature.

different warnings. Figure 4 depicts the integration with the code editor and the generated warnings.

V. User Study

We carried out an user study to validate the usefulness of the visualizations proposed in this paper. In particular, we thought to answer to these two research questions:

RQ1: Do the proposed visualizations efficiently aid the user to quickly find a fault?

RQ2: Is GZOLTAR a usable toolset?

This section also details the user study and draws conclusions from the results obtained from the user feedback.

A. Users and Setup

The user study was performed by 40 students (39 male and 1 female) on an iMac with OS X Mountain Lion version 10.8.3. All participants are students of the Master in Informatics and Computing Engineering in Faculty of Engineering of University of Porto.

All participants were familiar the Eclipse IDE and were divided into two groups according to the answers given in a questionnaire to assess their experience (in particular, regarding JUnit and Java). The users with JUnit experience were asked to perform the task without GZOLTAR (control group), whereas the others were asked to use GZoltar (test group). The size of each group (20 users) was based on the work by J. Nielsen's on usability and user tests [20], [21].

Before performing the debugging task, participants answered a questionnaire to guide the creation of the two groups. The users had no previous contact with the GZOLTAR toolset. Some of the users were not even familiar with the Eclipse IDE (the only IDE supported by GZOLTAR at the moment). Before starting the testing and debugging task, the toolset was not introduced in detail in order to let us to figure out how intuitive it is. Essentially, the main features were quickly introduced. The test group, also answered a questionnaire about GZOLTAR's usability.

To evaluate the efficiency of the visualizations, we used the XStream[4] project, a software to (de)serialize objects into eXtensible Markup Language (XML). None of the users were

[4]XStream homepage http://xstream.codehaus.org/, 2013.

familiar with the XStream's source code before the user study. XStream version 1.4.4 has 17389 Line Of Code (LOC), 306 classes and 22 packages. The program also provides 1418 unit test cases. We inject a logic operator fault in the program: a not equal to operator ("!=") was changed to an equal to operator ("==") in line 455 of the `AnnotationMapper` class from the `com.thoughtworks.xstream.mapper` package. This fault still allows the code to be compiled (a requirement to use GZOLTAR), and leads to unexpected behaviour during execution. Participants were provided with all test cases, and a timeout of 30 minutes was set to find and fix the fault.

B. The efficiency of visualizations

RQ1: Do the proposed visualizations efficiently aid the user to quickly find a fault?

From the test group, 100% of the participants found the fault and even fixed it. From the control group, only 35% found and fixed the fault, the other 65% did not even find the fault. For those who did not find the fault in 30 minutes (the timeout set by us), were assigned as taking the maximum time (the graphs and calculations presented later take this into account).

The control group found the fault in 23 minutes and 22 seconds (on average), with a median time of 30 minutes and a standard deviation of 9 minutes and 49 seconds. The test group found the fault in 7 minutes and 53 seconds (on average), with a median time of 7 minutes and 3 seconds and a standard deviation of 4 minutes and 52 seconds. From these results, we saw that those that used the visualizations were significantly faster in finding the injected fault (see Figure 8 and 9 for detailed information about the distribution of the groups).

From Figure 8, based on a kernel density estimation (KDE) [22], we see that the control group was more concentrated in the timeout value (30 minutes) and follows a bimodal distribution. We also identify two distinct subgroups, one with lower density and time values and another with higher density and time values. The test group was more concentrated in the lower time values and follows an unimodal distribution.

In Figure 9, we see that the median has significantly different values. Control group has a median value higher than test group. The highest concentration of participants is also different, with significant lower values at those used the visualizations.

To increase our certainty that the use of visualizations is of added value, we performed the statistical hypothesis *t-test* [23]. *T-test* is a statistical hypothesis test in which the test statistic follows a *Student's t* distribution if the null hypothesis is supported. Even thought the groups do not follow a *Student's t* distribution, the *t-test* is applicable, only without the precision of a perfect normal distribution. The null hypothesis of the *t-test* is used to determine if two groups are significantly different from each other. In our case the null hypothesis (H_0) tests if the mean time (μ_1) of the control group is less than

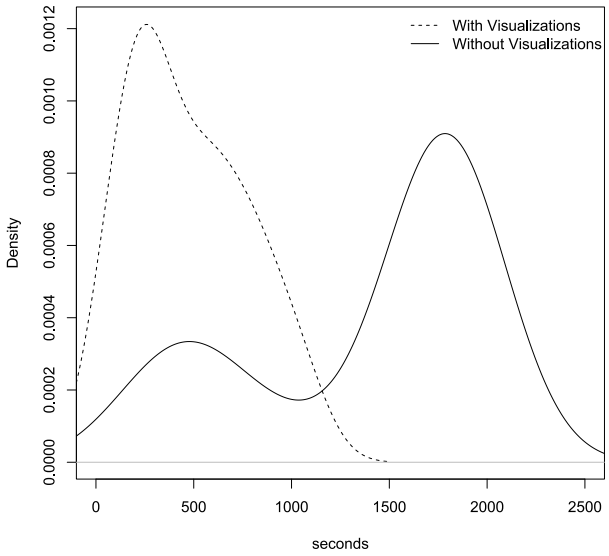


Figure 8: Times distribution.

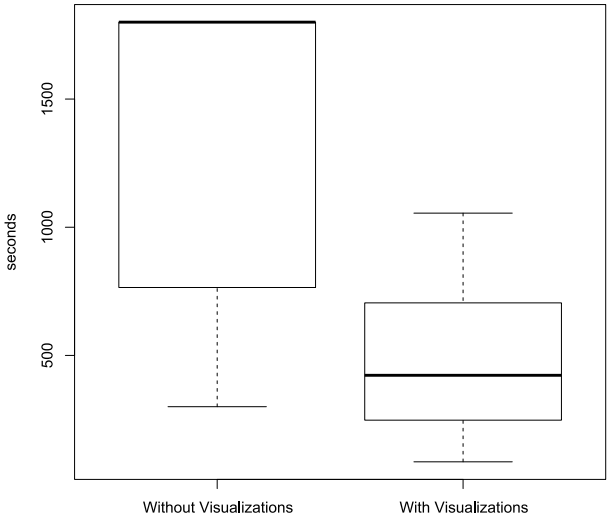


Figure 9: Time to find a fault.

or equal to the mean time (μ_2) of the test group. Formally, $H_0 : \mu_2 \geq \mu_1$.

Using *t-test* we can refute the null hypothesis: the t value calculated ($t \approx 6.163$) is greater than the *critical-t* $= 2.467$ (from *Student's t table* with 99% of confidence and 28 *degrees of freedom*). This means that the mean time spent by the control group to find the fault is greater than the mean time spent by the test group ($\mu_2 < \mu_1$). By refuting (H_0), we concluded that the visualizations help the debugging task.

To check how much faster was the test group against the control group, an alternative hypothesis $H_1 : \mu_2 + \Delta_x < \mu_1$ was proposed, where the Δ_x is the value of the average difference between performing the debugging task with and without using visualizations. We concluded that the test group took on average $\Delta_x = 9$ minutes and 17 seconds less than the control group to find the fault injected.

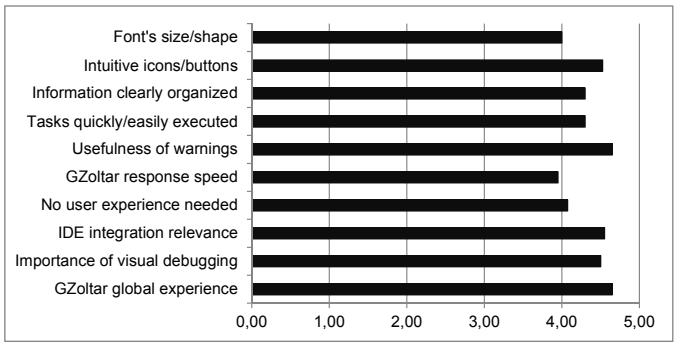


Figure 10: Average classification per topic of the GZOLTAR user's feedback.

To have a better understanding about the meaning of these values on finding real faults we scaled the timeout to: (1) 4 hours (half day's work), (2) 8 hours (day's work), (3) one week and (4) one month, and calculated again the value of Δ_x. The results were: (1) $\Delta_x \approx$ 1 hour and 30 minutes, (2) $\Delta_x \approx 3$ hours, (3) $\Delta_x \approx 2$ days and 16 hours and (4) $\Delta_x \approx 11$ days and 10 hours. With this we concluded that using visualizations to support software fault localization, developers are able to find a fault significantly faster than with common debugging tools.

C. The usability of GZOLTAR

RQ2: Is GZOLTAR a usable toolset?

After the debugging experiment, the test group was invited to answer a final set of questions. Most questions had the intention to ask for feedback about the user experience with respect to the GZOLTAR toolset. The questionnaire was comprised of 13 closed questions related to usability and intuitiveness of the GZOLTAR toolset. The answers were given with a Likert scale from 1 to 5, which 1 is nothing favourable and 5 very favourable (see Figure 10 to see the classification of each topic covered in the questionnaire). From their answers, we concluded that the vast majority of users found GZOLTAR an intuitive and helpful toolset.

Regarding usability, questions about font's size, information organization and highlights (colours, icons and buttons) were asked.

The average feedback about the highlights, and task execution intuitiveness and overall toolset performance was positive. The same happened when users were questioned about the experience and previous knowledge needed to use this toolset. And this corresponds exactly to the results of the user study, where 100% of the users were able to find and fix an unknown fault in less than 30 minutes, in a 17389 LOC software where they had no previous contact. This indicates that the toolset is shown to be effective.

An expressive number of users considered the toolset straightforward to start using and with a not too steep learning curve. Considering that the users had no real previous contact with the tool and that no advanced training in any form was given, we believe that these results are rather positive.

The users were also questioned about some concepts related to this toolset, such as the automatic debugging, integration of debugging into an IDE, visual debugging and about their global experience. The concept that had better receptiveness was visual debugging. These results also denote that the users approve visual automatic debugging tools integrated into an IDE.

In general, the GZOLTAR had a great acceptance among the users. The concepts underlying this toolset were also well accepted. These user reviews, added to the results of the study, reveal that the GZOLTAR toolset is efficient and effective.

Users were also invited to give their opinions and suggestions for further improvements. Sunburst was considered to be the most intuitive because, in the opinion of the participants, it represents the software hierarchy in an intuitive way. The collected feedback was overall positive, in which the users stated they found the toolset very effective and easy to use.

The suggestions to improve this toolset in future versions, were about simplifying the Bubble Hierarchy visualization to scale in a better way for big projects when we are using a small view.

With this experiment we were able to confirm the usefulness of this tool. The scenario of this experiment was rather demanding, because the users had no previous contact with the toolset or with XStream source code. Nevertheless, the results were very promising, and the users shown to be pleased with the use of this toolset.

Note that we have not explicitly asked which features of the toolset they used and/or found useful while searching the faulty statement because we have monitored the fault localization process. Since such monitoring allowed us to verify the procedures/steps taken by the users, we concluded that the tool was heavily relied upon. In fact, a user has told us that "without the toolset, it would be practically impossible to locate the fault".

Finally, we computed the correlation coefficient to verify if there is any connection between these topics. We found a strong correlation between:

- *intuitive icons/buttons* and *information clearly organized*,
- *intuitive icons/buttons* and *usefulness of warnings*,
- *intuitive icons/buttons* and *tasks quickly/easily executed*,
- *information clearly organized* and *IDE integration relevance*, and
- *no user experience needed* and GZOLTAR *global experience*.

The intuitive icons and buttons are responsible for the clarity of the information and for allowing quick and easy tasks. The users did not feel the need to have previous experience with the tool, they were able to perform the tasks without an in-depth knowledge of it. We concluded that the editor warnings are useful and intuitive for the users. So, the intuitive and clear interface is one of the most important aspects responsible for easy adoption by users.

VI. Related Work

In the software research and industry communities, and as software complexity grows over time, the relationship between visualizations and software projects is becoming more and more important [24]. This is also a fact in the testing and debugging phases of the software development life-cycle. Resulting from the interconnection between the software debugging process and visualizations, there are some interesting tools to explore further.

Data Display Debugger (DDD) [25] is a debugging tool which helps the user to find software failures. DDD analyses each software execution and provides a step-by-step visual trace of the execution instead of a global analysis of the entire system.

Tarantula [9] is a visual debugging tool for programs written in the C language and uses SFL. Each LOC is highlighted accordingly the probability calculated when Tarantula runs the test suits against the System Under Test (SUT). This highlight is made using a colour gradient from green to red. The green means the minimum suspiciousness and the red means the maximum suspiciousness. The analysis made by Tarantula considers the system as a whole, and this is a good feature when applied to big projects.

KDbg [26] is basically a front-end graphical user interface to GNU debugger (gdb). KDbg itself is not the debugger, it communicates with the gdb by sending commands to it and receiving the output. The main objective is to provide an intuitive interface for setting breakpoints, inspecting variables, and stepping through code.

EzUnit [13] is one more visual debugging tool which uses statistical debugging. More specifically is a plugin for the Eclipse IDE and works with projects written in Java language and which use JUnit test cases. EzUnit displays a list of the code blocks of the project analysed ranked by their failure probability. In this display, each line of the list is highlighted accordingly a colour gradient from green to red, where green is the minimum probability of failure and the red the maximum probability of failure. Beyond this, EzUnit has a call-graph related to all methods the test method potentially calls. Each node of the graph corresponds to a code block, and obeys to the same colour gradient.

VIDA [10] is a tool implemented as an Eclipse IDE plugin. It supports programs written in Java language and JUnit test cases. At first VIDA collects the statement coverage information of test cases and select a failed test that has executed some statements, because this test has some suspicious statements to be examined. This selected test is the starting point of the debugging process. After that, the objective is to locate the faulty statement that caused the failure of that test. VIDA takes into account the statements with large suspicious and helps visually the user to set a breakpoint, giving some candidates. After the setting of the breakpoint, VIDA collects the user's estimation on that breakpoint and modifies the statements' suspicious based on the user's estimation.

Enhancing Fault Localization via Multivariate Visualization approach [11] uses scatter plots to represent the similarity between test cases. The main goal is facilitate the identification of similar tests. The similar test are represented close to each other and the dissimilar tests are represented away from each other. Before the scatter is built, it is necessary to calculate some auxiliary matrices. In the initial matrix, each line is a test case and each column is an element of the program inspected. The second matrix is calculated applying a dissimilarity metric that calculates the level of similarity between the test cases. Finally is calculated the third matrix from the previous using a multidimensional scaling [27] to be possible a 2D representation of the test cases as scatter points. The user is helped to find the coincidentally correct tests by following some scatter plots. These coincidentally correct tests are moved to the set of failure tests or discarded to enhance the effectiveness of Coverage-based Fault Localization (CBFL) (another term to refer to SFL) [12].

Java Interactive Visualization Environment (JIVE) [28] is based upon an Eclipse IDE plugin architecture and the Java Platform Debugging Architecture (JPDA) is its key component. The aim of this tool is to provide a more comprehensive declarative and visual execution environment. JIVE uses JPDA's Java Debug Interface (JDI) to request notification of certain runtime events. The information collected from the call stack together with the JDI event forms a JIVE event. JIVE constructs two main models, an object model and a sequence model. The object model represents the program's execution state and the sequence model details its history of execution. The objective is to facilitate the program comprehension and debugging. This tool has three main aspects: (1) scalable visualizations - it supports on the object diagrams suppressing of the internal details of objects and their interactions, suppressing superclass details, hiding field tables, showing only objects involved in the class path, etc. On sequence diagrams, the objective is to reduce them. Sub-computations corresponding to large call threes are replaced by a single node; (2) declarative queries - here the aim is select only those parts of the object and sequence diagrams that are really important to a query of interest to the programmer. So, the sequence diagram is an good way to visually reporting the answers to queries, helping the identification of where answers lie; and (3) interactive execution - JIVE does incremental state-saving during forward execution and incremental state-restoring during reverse execution.

We are distinct because the GZoltar toolset offers a great level of interaction for the user. Our interactive graphical visualizations obey to an hierarchical structure and use an intuitive colour gradient to represent the suspiciousness value of each component. The fact that GZoltar is directly embedded on Eclipse IDE, provides a faster an easier inspection of the code, improving the debugging process.

VII. Conclusions and Future Work

Automatic fault localization has been an active area of research in recent years. Researchers have, however, focused more on how to improve the diagnostic quality, and far less on how to show the information to the end users (i.e.,

developers). The latter may have a tremendous impact in fault localization [7]. In the past, we have exploited OpenGL as basis to build interactive visualizations, but these techniques entail technological drawbacks: the complex scenes rendered could be very heavy in terms of computational resources used, entailing a negative impact in the user interaction with the tool; there are incompatibilities with some GPU's and graphics cards drivers originating a defective rendered scene.

In this paper, we proposed visualizations using HTML5 to display the diagnostic reports yielded by automatic fault localization techniques. In particular, we consider as underlying technique the Ochiai spectrum-based fault localization [19], known to be amongst the best performing techniques [4]. We offered the proposed visualizations within the GZOLTAR toolset, an Eclipse plugin for testing and debugging. The visualizations explored by us are the Sunburst, Vertical Partition, and Bubble Hierarchy.

To ascertain the usefulness of visualizations, we carried out an user study. Results confirmed that without using visualizations to aid the fault localization is too much difficult to find a fault, only 35% found it, and even when the fault was found, users spent significantly more time to do it. The group that used the visualizations took on average 9 minutes and 17 seconds less than the group that did not use them. The intuitive and clear interface of the visualizations is one of the most responsible for this times difference.

Future work includes the following. We plan to perform an user study with more participants, ideally from industry. Second, we plan to offer the plugin to more IDEs (such as IntelliJ IDEA). Finally, other visualizations will also be explored.

ACKNOWLEDGMENTS

This work is financed by the ERDF – European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT – Fundação para a Ciência e Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020484. We would like to thank Nuno Cardoso, for the useful feedback on previous versions of this paper.

REFERENCES

[1] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, Jan. 2002.

[2] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: An Eclipse Plug-in for Testing and Debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 378–381.

[3] E. Burnette, *Eclipse IDE Pocket Guide*. O'Reilly Media, Inc., 2005.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, Nov. 2009.

[5] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, ser. PRDC '06. Riverside, California, USA: IEEE, 2006.

[6] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-Driven Documents," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2301–2309, Dec. 2011.

[7] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209.

[8] T. Janssen, R. Abreu, and A. J. C. v. Gemund, "Zoltar: A Toolset for Automatic Fault Localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009.

[9] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 467–477.

[10] D. Hao, L. Zhang, L. Zhang, J. Sun, and H. Mei, "VIDA: Visual interactive debugging," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 583–586.

[11] W. Masri, R. A. Assi, F. Zaraket, and N. Fatairi, "Enhancing fault localization via multivariate visualization," in *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ser. ICST '12. USA: IEEE, 2012, pp. 737–741.

[12] W. Masri and R. A. Assi, "Cleansing Test Suites from Coincidental Correctness to Enhance Fault-Localization," in *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. IEEE Computer Society, 2010, pp. 165–174.

[13] P. Bouillon, J. Krinke, N. Meyer, and F. Steimann, "Ezunit: a framework for associating failed unit tests with potential programming errors," in *Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming*, ser. XP'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 101–104.

[14] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, Jan. 2004.

[15] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," in *Proc. of the 6th European Software Engineering conference co-located with the 5th ACM SIGSOFT international conference ESEC/FSE*, ser. ESEC '97/FSE-5. Springer-Verlag New York, 1997, pp. 432–449.

[16] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An Empirical Investigation of the Relationship Between Fault-Revealing Test Behavior and Differences in Program Spectra," *STVR Journal of Software Testing, Verification, and Reliability*, no. 3, pp. 171–194, September 2000.

[17] Q. Yang, J. J. Li, and D. Weiss, "A survey of coverage based testing tools," in *Proceedings of the 2006 international workshop on Automation of software test*, ser. AST '06. New York, NY, USA: ACM, 2006.

[18] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[19] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 89–98.

[20] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, ser. CHI '93. New York, NY, USA: ACM, 1993, pp. 206–213.

[21] J. Nielsen. (2012) How many test users in a usability study. http://www.nngroup.com/articles/how-many-test-users/.

[22] E. Parzen, "On Estimation of a Probability Density Function and Mode," *The Annals of Mathematical Statistics*, vol. 33, no. 3, 1962.

[23] M. Hazewinkel, *Encyclopaedia of Mathematics*, ser. Encyclopaedia of Mathematics. Springer, 1994.

[24] S. Diehl, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[25] A. Zeller and D. Lütkehaus, "DDD - A Free Graphical Front-End for UNIX Debuggers," *ACM SIGPLAN Notices*, vol. 31, no. 1, 1996.

[26] J. Sixt. (2011) KDbg - A Graphical Debugger Interface. http://www.kdbg.org/.

[27] I. Borg and P. Groenen, *Modern Multidimensional Scaling: Theory and Applications*, ser. Springer Series in Statistics. Springer, 2005.

[28] J. K. Czyz and B. Jayaraman, "Declarative and visual debugging in eclipse," in *Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, ser. eclipse '07. New York, NY, USA: ACM, 2007, pp. 31–35.