On the use of a multiple-visualization approach to manage software bugs

Renato L. Novais¹,², Manoel Mendonça¹, Diogo Maron¹, Ivan do Carmo Machado¹, Caio A. N. Lima²

¹Computer Science Department, Federal University of Bahia (UFBA), Bahia, Brazil
²Information Technology Department, Federal Institute of Bahia, Campus Santo Amaro, Bahia, Brazil

{renatoln, mgmendonca, diogomaron, ivanmachado}@dcc.ufba.br, ticaolima@gmail.com

Abstract. Software testing has been reported as one of the main responsible for quality assurance in a software project. In this effect, a series of techniques, methods, and tools can be found in the literature. In this work, we propose an approach to manage bugs using multiple software visualization. We developed an eclipse-based plug-in to report bugs. As a result of this activity, the collected bugs may be bound with the software elements. Such a binding aids the engineers responsible for fixing bugs to know the exact point where the reported bug was installed. We used multiple views with different perspectives to analyze and understand the bugs in the source code. As a proof of concept, we simulated its use in an open source system.

1. Introduction

One of the main goals of Software Engineering is to study techniques, methods, and tools that yield high quality software artifacts, i.e. software with as few faults as possible [Endres and Rombach 2003].

There exist several systems to manage software faults, the so-called Bug Tracking Systems (BTS), such as Bugzilla¹, Mantis² and Trac³. Those BTS are generally web-based and can be used by different stakeholders to report and manage software issues. Some typical use cases for BTS are: i) final users can report functional faults discovered during the use of a system - the users usually do not know how the software is implemented or what software entities are causing the bug; ii) developers can, during the development phase, identify bugs and flag it to be fixed later; iii) inspectors can identify bugs and improvement opportunities and report them in the BTS.

In the latter two cases, the reporters may know where the bugs or improvement opportunities are in the code. This may be considered as a valuable information, since fault localization is usually a costly process [Hamill et al. 2009]. However, current BTS do not have a specific field to record this information. Moreover, they are usually not integrated to the source code where the fault was indeed pinpointed, not using for

¹ Bugzilla website: http://www.bugzilla.org/
² Mantis website: http://www.mantisbt.org/
³ Trac website: http://trac.edgewall.org/
example, an IDE. Thus, the reporter has to count on text notes to describe the fault and identify the software entities responsible for the bug in the code. However, text notes are difficult to be traceable.

The first part of our work is focused on bridging this gap. We have worked on a software solution that enables the reports (related to ii and iii, aforementioned) of bug issues during the detection phase, linking it to the associated software(s) entity(ies). An Eclipse-based plug-in was created that enables the user to report a bug in this same IDE. In addition, we also handle issues regarding test process, e.g. how to manage the reported bugs. In this sense, we deal with questions such as: (a) Who will fix the bug? (b) What is the bug severity? (c) How are the bugs spread in the source code?, etc. Management will help the user to define the best strategies to fix the bugs. This kind of information is already handled by existing BTSs, but additional (and extremely important) information is not, e.g. a developer wants to know how the bugs are dispersed in the source code, or even, what component of your software is more affected by the bugs? or, if a bug is associated with a component X and X is coupled with Y, what is the impact of this bug in Y?

The second and main part of our work addresses such open questions. To reach these goals, we propose the use of multiple-perspective software visualization approach [Carneiro et al. 2010b]. During the past recent years, we have been developed a multiple-perspective software visualization environment named SourceMiner [Carneiro et al. 2010a]. This is a general-purpose software visualization environment that is integrated with Eclipse IDE. It provides several different integrated views to visualize Java-based software projects.

The work presented in this paper elaborates on SourceMiner with software bugs information. The current work uses three visualization metaphors to present the bug information from three different perspectives: structure, inheritance and dependency. A software layer was developed to access information directly from the Mantis BTS.

This paper presents the approach, describes the resources provided by SourceMiner for its support, and discusses ways of using them for bug management analysis. The remainder of the paper is organized as follows. Section 2 discusses our approach. Section 3 shows the approach in action. Section 4 discusses related works. And, Section 5 concludes the papers with an outlook at future work.

2. The approach

The approach is comprised of the steps presented in Figure 1. As aforementioned, developers can, while developing, identify and report bugs to be fixed later. Likewise, inspectors can identify bugs and improvement opportunities in the inspection phase. All the collected information is stored in a BTS (Figure 1 - top). To this end, it is necessary the use of a BTS integrated to the IDE (Section 2.1).

Once the information is collected, we propose the use of a multiple-perspective visualization to manage the bugs (Figure 1 - bottom). We focus on the use of three

---

4 http://www.sourceminer.org/
5 by metaphor we point to [Diehl, 2007]
views: Treemap, Polymetric and Coupling (Section 2.2). Both developers and managers may benefit from the given visualization.

![Diagram](image.png)

**Figure 1. The proposed approach diagram.**

### 2.1. Reporting bugs inside an IDE

By using the Eclipse plug-in, the reporter can report bugs inside the IDE. The plug-in provides interaction over classes or methods in the package explorer view. With a right-click mouse interaction, the user selects the class/method where the bug was identified and uses the interaction. After that, the user is required to fill some information about the bug. Thus, the bug is stored in a BTS bound with the software element.

The current plug-in version uses the Mantis BTS. Mantis was chosen because it is one of the most popular BTS used in the open source community, and it is possible to extend its database with new customized fields.

For the purpose of the plug-in development, two customized fields were included: *(i)* **Method** - store bugs associated to Java methods; *(ii)* **Type** - store bugs associated to Java classes. Using the plug-in, the user can fill all other information about the bug that is required by Mantis BTS, such as: summary, description, assigned to, severity, etc. The user can even associate the current detected bug with a previous detected one, since the same bug can be scattered over different methods or classes.

The information of the software elements affected by the bug is stored in customized fields using XML-based tags. Figure 2 shows an excerpt of a stored bug. Figure 2a is the syntax used for bugs affecting classes, and Figure 2b is the syntax used for bugs affecting methods. The tags used are: `<type>` to represent the class; `<method>` to represent the method; `<name>` to represent the name of the method; and `<signature>` to represent the signature of the method.

By analyzing the aforementioned scenario, we can notice two gains in the proposed approach: *(i)* efficiency in the inspection process – the users do not need to use more than one program to perform the inspection and to store the found bugs (BTS stand alone application and an IDE). It can be performed in the same platform.
Therefore, the inspector does not lose time interchanging over the two applications, for instance; (iii) more precise information about where the bug is installed – the application provides the linking between the errors and the place, in terms of methods and classes, enabling an easier way to find what artifacts are affected by a specific bug.

![XML tags used to represent a bug.](image)

Figure 2 Sample of XML tags used to represent a bug.

This valuable information can be used for several purposes. Pareto's Law [English et al. 2009] says that, for many events, around 80% of the effects come from 20% of the causes. This principle can also be used for software test [Thomas et al. 2005][Pressman 2005]. The problem is to discover which modules of the software represent the 20%. Our approach supports this identification, using the bind of the bug to the software element and using Software Visualization. As a consequence, some advantages may be identified: once these modules are known, the software test effort can be directed to these modules; it may reduce costs associated to the test phase; the quality of the software can be improved, etc. In this work, this information is also used for the management of bug using multiple views software visualization.

### 2.2. Managing bugs with Multiple Views Software Visualization

Software visualization (SoftVis) is a field of software engineering that aims to help people to understand software through the use of visual resources [Diehl, 2007]. SoftVis has been used with different purposes, e.g. comprehension and evolution. This work uses SoftVis to analyze and understand bugs in a software system.

To better analyze the bugs inside the software we propose the use of multiple perspective software visualization [Carneiro et al. 2010b]. Three views with different metaphors are used, as shown in Figure 1. Each view portrays a different perspective of the software. This is desired since the use of multiple perspectives improves the software comprehension.

Besides, the use of a color-based approach is encouraged, since this visual resource aids in the identification of important elements. Hence, our proposed approach has chosen the red color to decorate software elements affected by bugs. The brightest red elements are the ones with more bugs associated, representing the hot spots in the system.

As a generic data exploration strategy, we recommend the use of the treemap view to understand the big picture of the bugs spread in the system. In this structural view, rectangles representing methods of the same class are drawn together inside the rectangle of the class. Likewise, the rectangles of the classes that belong to the same package are drawn together inside the rectangle of the package. Using this view, one can visually, and easily, identify the modules with the larger amount of bugs in the systems, and so, define strategies of what modules should be tested first or most intensively.
The *Polymetric* view portrays other perspective of interest in the OO systems: inheritance. It is important to visually show which classes extends others or implement certain interfaces. The bugs are now represented in an inheritance perspective. We argue that bugs, regardless the information reported by user in the BTS fields (e.g severity or priority), on the top of an inheritance are more harmful than the bottom ones. In this scenario, the bugs on the top are probably affecting the derived classes.

The *Coupling* view shows the afferent and efferent coupling between classes. With this view, one can analyze the impact of an element in their dependents, or in the elements it depends on. In this case, also without consider the severity/priority reported by the user, the view allows a more interesting analysis, such as: it is possible to infer that bugs in classes with highest afferent coupling are more harmful than the others, since more classes depend on the buggy one.

In addition, the plug-in implements the use of filters. It helps the user to filter the views with the BTS information such as: severity, priority, all or open bugs, etc.

In general, the importance of the bug is not only about which is reported by the user in the severity/priority field, for instance, but also the importance inherent to the visualization metaphor.

### 3. An example of use

In order to analyze the viability of using our proposed approach, we gathered data from the open source project Demoiselle. We accessed the project repository and extracted all the commits posted from January 28th up to May 25th, in 2011. We dumped the repository data into a file, which contained the amount of 2624 changed paths.

A filter was applied in the collected data to get only commits related to bugs and java classes from the *Demoiselle Framework core*. The final scenario was composed of 182 changed paths. We associated every change path to a bug entry in the Mantis instance, binding it to the java classes modified in the commit. Although our approach supports the handling of methods and classes, we only considered classes in this study, since this initial focus was to verify if the results were convincing.

The views portraying the bug information gathered from the *Demoiselle Framework core* are presented in Figure 3.

The *Treemap view* (Figure 3 Top) shows the bugs’ big picture of the project. The class with highest number of bug is easily identifiable. `SecurityContextImpl` class has the brighter red color. Other classes are highlighted in the figure. They also have a high number of bugs (dark red color). It was possible to identify using this view packages with all classes bugged. As an example we can cite the package `br.gov.frameworkdemoiselle.security`, which contains the highlighted classes `SecurityContext` and `Authorization`, and the package `br.gov.frameworkdemoiselle.internal.configuration`, which contains the highlighted class `SecurityConfig`.

The Polymetric view (Figure 3 bottom-left) portrays the bugs in an inheritance perspective. Two classes (`ConfigurationException` and `ConfigurationBootStrap`) have no bugs, instead of extends a buggy class. Perhaps the inherited classes may negatively impact

---

6 Demoiselle Framework website: http://www.frameworkdemoiselle.gov.br/
such classes. Alike the Treemap representation, SecurityContextImpl class has the brighter red color. The biggest class in the picture is ConfigurationProcess. SourceMiner draws the length of the classes according to the number of lines of code. This class has a relative high number of bugs.

The Coupling view (Figure 3 bottom-right) portrays the bugs in a dependency perspective. We filtered the view to show only the afferent (fan-in) coupling of the classes. It is possible to identify classes that have no bugs, but use (depends on) a buggy class. As an example, we have the classes: CoreBundle, PaginationContextImpl, DelegateCrud, Strings, Contexts, DefaultMessage and ConfigurationBootstrap. One interesting observation in this view is that three classes (CoreBundle, DelegateCrud, DefaultMessage) with no bugs are using the same buggy class (Bean class).

Figure 3. Views with bugs on Demoiselle framework core project

The use of multiple views software visualization to manage software bugs has contributed to the identification of hot spots in the source code of this analyzed system. The study showed that the approach brings important gains to managers and users.

4. Related Work

Due to the importance of software testing, several authors have proposed different techniques to deal with software test management. We selected some relevant studies that are in line with our ideas. These are next described.

Jones (2004) proposed an approach aimed at selecting suspicious statements that may contain the fault. The technique uses color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases [Jones 2004]. According to the author, based on this visual mapping, a user can inspect the statements in the program, and
identify potential faulty statements. Lenon et al. (2000) explore the use of multivariate visualization techniques to support an observation-based testing approach. The work seeks to identify and compare execution profiles that have failed at some point. In this way, executions with the same profile tend to have the same defect [Lenon et al. 2000].

End-User software visualizations for fault location were proposed by Ruthruff et al. (2003). They worked on a vision they call “end-user software engineering,” to help solving the problem of pervasive faults in end-user programming. They focused their ideas in the spreadsheet paradigm, since it is so widespread in practice. The work shows several fault localization visualization techniques for spreadsheets bringing the benefits of software visualization to end-user programmers. These works do not use the concept of multiple views addressing different perspectives as our work does. We believe it can improve the way to manage bugs.

The closer approach we found was proposed by Dambros et al. (2007). They present two visualization techniques aimed at understanding bugs at two different levels of granularity: System Radiography – this visualization renders bug information at the system level and provides indications about which parts of the system are affected by what kind of bugs at which point in time; Bug Watch – This visualization provides information about a specific bug supporting the characterization of bugs and the identification of the most critical ones [Dambros et al. 2007]. In our approach we do not deal with the bug’s life, however, differently from this work, we associate bugs information with structural, inheritance and dependency properties of the software.

The majority of related works address the fault detection/Location. We address a different point: the management of the already reported/detected/localized bugs. In order to accomplish such a goal, we use the multiple-perspective views approach plugged in an IDE and in a BTS in our proposal.

5. Concluding Remarks and Future Work

In this work, we proposed an approach to manage bugs using multiple software visualization. We developed a plug-in to report bugs inside the Eclipse IDE. The bugs analyzed were stored in a Mantis BTS instance. Bugs storage is an activity that may be performed during software inspection tasks. We proposed that, as a result of this activity, the collected bugs could be bound with the software elements. This binding aids the responsible for fixing bugs to know the exact point where the reported bug was installed. We used multiple views with different perspectives to analyze and understand the bugs in the source code.

Managers can benefit from these views when taking important decisions about the software test process.

The proposed approach still requires a series of improvements. It currently only uses the Mantis BTS, which limits the range of projects that can benefit from it. As a future work, we intend to use the Bugzilla BTS. As another direction, we are planning to use the approach in real projects, so that we could gather more confidence about the results. In this sense, a structured and formalized experimental study would be set.

Acknowledgments. This work was partially supported by the National Institute of Science and Technology for Software Engineering (INES - http://www.ines.org.br/), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08.
References


