BugMaps: A Tool for the Visual Exploration and Analysis of Bugs

Andre Hora, Nicolas Anquetil,
Stephane Ducasse, Muhammad Bhatti
RMoD Team
INRIA, Lille, France
{firstName.lastName}@inria.fr

Cesar Couto
Department of Computing
CEFET-MG, Belo Horizonte, Brazil
cesarfmc@dcc.ufmg.br

Marco Tulio Valente, Julio Martins
Department of Computer Science
UFMG, Belo Horizonte, Brazil
{mtov,jleandro}@dcc.ufmg.br

***Abstract*—To harness the complexity of big legacy software, software engineering tools need more and more information on these systems. This information may come from analysis of the source code, study of execution traces, computing of metrics, etc. One source of information received less attention than source code: the bugs on the system. Little is known about the evolutionary behavior, lifetime, distribution, and stability of bugs. In this paper, we propose to consider bugs as first class entities and a useful source of information that can answer such topics. Such analysis is inherently complex, because bugs are intangible, invisible, and difficult to be traced. Therefore, our tool extracts information about bugs from bug tracking systems, link this information to other software artifacts, and explore interactive visualizations of bugs that we call bug maps.**

I. Introduction

Currently there are a number of tools for software analysis [1], [2], [3]. Such tools use different types of information about the structure and history of a system. Basically, these tools are used to analyze software evolution, manage the quality of the source code, compute metrics, analyze coding rules, etc. In a general way, these tools help software engineers to understand large amounts of data that come from software repositories.

On the other hand, one source of information has been less explored by existing software analysis tools: the bugs on the system. Some tools already analyze such information [4], [5], [6], but little is known about the evolutionary behavior, lifetime, distribution, and stability of bugs. Moreover, reasoning about bugs is a task inherently complex, because bugs are intangible, invisible and difficult to be traced. Particularly, such analysis is complex because it involves: (i) retrieval of data from bug-tracking and version control platforms; (ii) mapping of bugs to defects in software modules; and (iii) data processing to extract and reason about relevant information.

In this paper, we present the BugMaps tool that provides mechanisms to automate the process of retrieving and parsing software repositories data, algorithms to map bugs reported in bug-tracking platforms to defects in the classes of object-oriented systems and that provides visualizations for decision support. More specifically, the tool has the following features:

- The tool automatically extracts a time series with number of defects at the class level from information available in bug-tracking and version control platforms.
- The tool integrates models extracted from the source code with the number of defects time series.
- From this integration, the tool provides a set of interactive visualizations that supports software developers and managers in answering questions such as: (a) What are the modules involved in bug-fixing? (b) What is the lifetime of a bug? (c) What is the period that a module has presented more bugs? (d) What modules are stable or unstable with respect to bugs? (e) What are the modules whose number of bugs has increased or decreased over time? (f) What is the total number of bugs of a module?

The paper is organized as follows. In Section 2 we introduce BugMaps, using illustrative examples extracted from the bugs reported for the Eclipse JDT system. In Section 3 we discuss related work, and in Section 4 we conclude the paper.

II. BugMaps

Figure 1 shows the architecture of the BugMaps[1], which includes the following components:

1) **Mapping Module.** This module receives as input the log files from version control platforms – CVS or SVN – and the bug reports from bug-tracking platforms – Jira or Bugzilla. This module maps bugs to defects in classes and creates the times series number of defects (i.e., for each class, a time series that provides the number of defects in a given time frame).
2) **Visualization Module.** This module receives as input the series number of defects, the models extracted from several versions of the source code and the source code itself. From this information, this module computes measures on bugs and provides many interactive visualizations.

A. Mapping Module

To create the time series of defects, we implemented an XML parser that reads the information provided by the CVS/SVN repositories and extracts the developer's comments and the changed classes. Then, another XML parser reads the bug reports available in the Jira/Bugzilla repositories and collects the date on which each bug was reported and its

[1] http://rmod.lille.inria.fr/web/pier/software/BugMaps

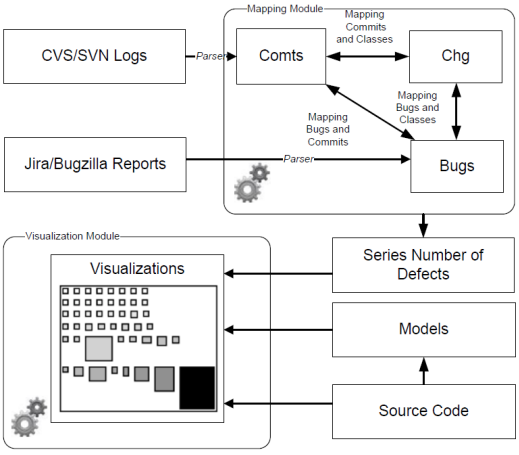

Fig. 1. BugMaps architecture

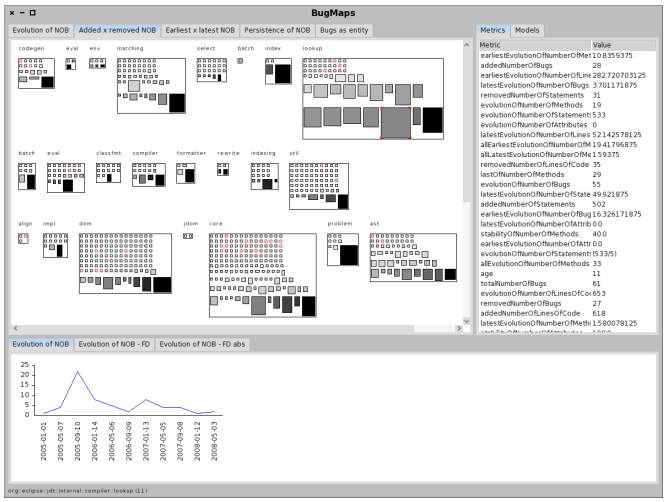

Fig. 2. BugMaps: history browser

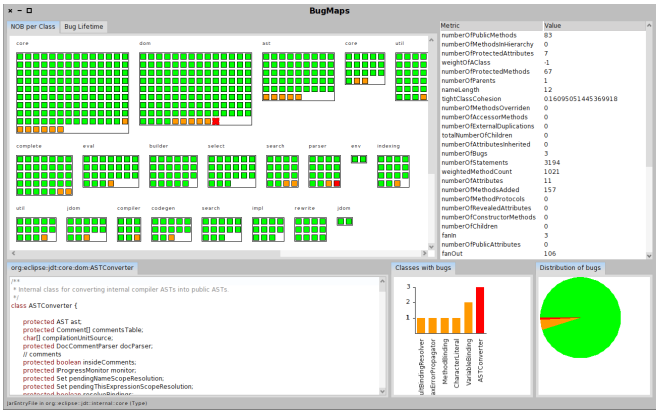

Fig. 3. BugMaps: snapshot browser

identifier. After that, we linked each bug b to the classes changed to fix b. More details can be checked in [7].

B. Visualization Module

This module receives as input the series of defects, models of the source code and the source code itself of the system under analysis. Models of source code are generated using VerveineJ parser[2]. Two browsers are then used for analysis, one to deal with the history of bugs (called history browser, which receives as input a history model [8]) and other to deal with a particular snapshot of the system under analysis (called snapshot browser, which receives as input a snapshot model). These browsers are implemented in the Moose Platform[3].

Figure 2 shows the history browser which is composed by three panes: visualizations (top left), measures (top right) and charts (bottom). The *charts* pane shows the number of bugs presented in a class/package during its timelife and the *measures* pane shows class/package measures, which are updated according to the selected entity in the *visualizations* pane. The visualizations displayed can be swapped using tabs presented in the top of *visualizations* pane. The history model of the source code is a collection of snapshot models, then from the history browser it is possible to open snapshot browsers using tabs presented in the top of *measures* pane.

Figure 3 shows the snapshot browser which is composed by four panes: visualizations (top left), measures (top right), source code (bottom left) and charts (bottom right). *Measures*, *source code* and *charts* panes are also updated according to the selected entity in the *visualizations* pane. The visualizations displayed can be swapped using tabs presented in the top of *visualizations* pane.

The next subsections detail the measures and visualizations.

1) Measuring Bugs History: We provide six measures to summarize the evolution of the bugs in a system. These measures are instantiation of *Evolution of a Version Property*, a generic evolution measure proposed by Girba and Ducasse [8].

[2]http://www.moosetechnology.org/tools/verveinej
[3]http://www.moosetechnology.org

The provided measures rely on a basic metric called ENOB_i, which is defined as the difference in the number of bugs (NOB) between version i and $i-1$ of the class C: $ENOB_i(C) = NOB_i(C) - NOB_{i-1}(C)$, for $i>1$. This basic metric is used to build six more advanced ones. We first define each advanced metric, before giving examples and intuition on their use.

Evolution of Number of Bugs (ENOB). $\text{ENOB}_{j..k}$ is the sum of the number of bugs added or removed from version j to version k: $ENOB_{j..k}(C) = \sum_{i=j+1}^{k} |ENOB_i(C)|$.

Latest Evolution of Number of Bugs (LENOB). $\text{LENOB}_{j..k}$ favors the recent changes (closer to the last version of the history) over the changes further in the past by applying a weighting function: $LENOB_{j..k}(C) = \sum_{i=j+1}^{k} |ENOB_i(C) * 2^{i-k}|$.

Earliest Evolution of Number of Bugs (EENOB). $\text{EENOB}_{j..k}$ favors the old changes (closer to the first version of the history) over the changes near the end of the experiment: $EENOB_{j..k}(C) = \sum_{i=j+1}^{k} |ENOB_i(C) * 2^{j-i+1}|$.

Added Number of Bugs (ANOB). $\text{ANOB}_{j..k}$ is the sum of the number of bugs added in the subsequent versions: $ANOB_{j..k}(C) = \sum_{i=j+1}^{k} ENOB_i(C)$ if $NOB_i(C) - NOB_{i-1}(C) > 0$.

Removed Number of Bugs (RNOB). $\text{RNOB}_{j..k}$ is the sum of the number of bugs removed in the subsequent versions: $RNOB_{j..k}(C) = \sum_{i=j+1}^{k} |ENOB_i(C)|$ if $NOB_i(C) - NOB_{i-1}(C) < 0$.

Bugs Persistence (BP). $\text{BP}_{j..k}$ is the number of versions from version j to version k containing at least one bug: $BP_{j..k}(C) = \sum_{i=j}^{k} 1$ if $NOB_i(C) > 0$.

2) Visualization: The visualizations provided by BugMaps are based on Distribution Map, a generic technique to reason about the result of software analysis and to help to understand how a given phenomenon is distributed across a software system [9]. Using Distribution Map three metrics can be displayed through height, width and color of the objects. In our maps, small rectangles represent class histories, bugs, or classes and containers represent packages or package history. BugMaps provides five maps based on the history of the bugs of a system (Figures 4-8) and two maps for a particular snapshot of a system (Figures 9-10).

We analyzed the Eclipse JDT system according to the proposed visualizations, which are showed in the next figures. It was collected 91 versions from 2005-01-01 to 2008-06-14.

Evolution of NOB. In this map, the height of a class is the *Evolution of Number of Bugs* measure and the color is the total number of bugs in the class lifetime. Therefore, the longer is the height of a class, the higher is the number of bug changes performed during its lifetime. In Figure 4, we can see that in package *lookup* about half of the classes are involved with bug changes and about half of the classes are free of bugs, which means that this package should have a special attention during the development.

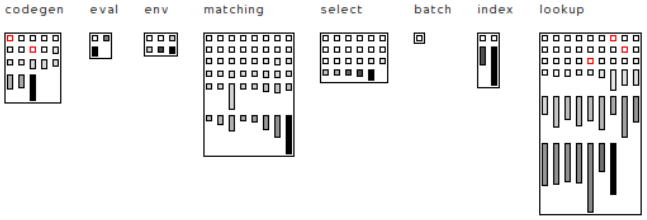

Fig. 4. Evolution of NOB

Added x Removed NOB. In this map, the height of a class is the *added number of bugs* measure, the width is the *removed number of bugs* measure, and the color is the total number of bugs during its lifetime. Therefore, if a class is similar to a square, it means that added bugs have also been removed. If a class has more height than width, it means that bugs have been more added than fixed. If a class has more width than height, it means that bugs have been more fixed than added, in the time period under analysis. This may happen if the period considered is not at the start of the system life, and there was bugs already identified but not corrected. In Figure 5, we can see that most of the classes that changed their number of bugs are square-shaped, which means that added bugs have also been fixed during the lifetime of the class.

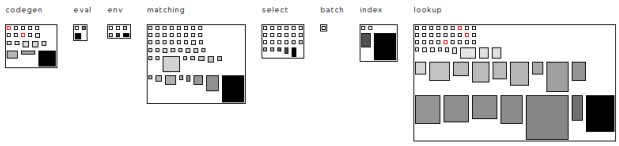

Fig. 5. Added x removed NOB

Earliest x Latest NOB. In this map, the height of a class is the *earliest number of bugs* measure, the width is the *latest number of bugs* measure, and the color is the total number of bugs during its lifetime. Therefore, if a class has more height than width, it means that bugs are closer to the first version under analysis (old bugs). If a class has more width than height, it means that bugs are closer to the last version under analysis (recent bugs). Figure 6 shows that bugs can be either close to the first (vertical shapes) and last version (horizontal shapes), which means that the bugs reported for such classes have been fixed during all the time frame of the experiment.

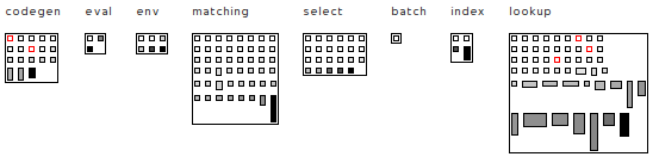

Fig. 6. Earliest x latest NOB

Persistence of NOB. In this map, the color of a class represents the *persistence of bugs* measure. Green means that there are bugs in less than 20% of the versions, orange means that there are bugs in 20% to 80% of the versions, and black means that there are bugs in 80% or more of the versions. White means that there are no bugs. In Figure 7, we can see that in package *lookup* bugs persistence is a problem, since there are several black classes, which means that bugs are persistent during almost classes lifetime.

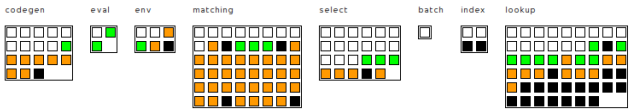

Fig. 7. Persistence of NOB

Bug as entity. This map represents bugs instead of classes. The color of a bug represents its *lifetime*, i.e., the number of days it stayed opened. Blue denotes a bug that was still opened at the end of the time period considered. White denotes a bug that was opened for a short time, going to yellow is a bug that was opened up to 3 months, and going to red is a bug that was opened for more than 3 months. The width of a bug representation denotes the bug complexity, measured as the number of classes changed to fix the bug. Bugs are sorted according to the date they were created. In Figure 8, complex bugs (long width) are dispersed in time,

which may mean that the system is not becoming so complex (bugs are spread all over it). Bugs going to red are also dispersed in time, which means that the developers are not spending more and more time solving bugs. There are many blue (opened) bugs at the end, and a few in the beginning.

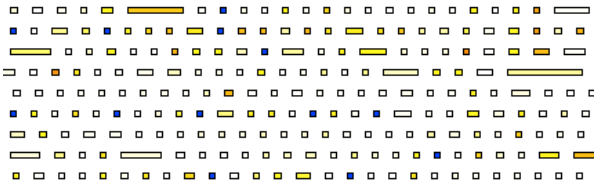

Fig. 8. Bug as entity

The BugMaps tool also provides the following maps for a particular snapshot of the system (i.e. maps that are not based on the history of versions):

NOB per Class. In this map, the color of a class represents the *number of bugs* in a particular version. Green means that a class has no bugs. Orange means that a class has one or two bugs. Red means that a class has three or more bugs. Therefore, this visualization provides an overview of the distribution of the bugs in a given snapshot of the system. Figure 9 provides an overview of the distribution of the bugs in one of the first versions of the experiment where we can see a small number of classes with bugs (orange/red).

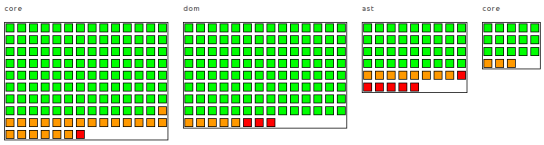

Fig. 9. NOB per Class

Bug Lifetime. In this map, the color represents the median *lifetime of the bugs* affecting a class. Green means that a class has no bugs or on median it took less than a week to fix its bugs. Orange means that on median it took between a week and a month to fix its bugs. Finally, red means that on median it took more than a month to fix its bugs. We consider the median because it is common to have bugs that last for years in the system, which bias the average. In Figure 10, we can see that there is no default behavior for the lifetime of the bugs affecting a class. There are classes that on median it took between one week and one month to fix the bugs and there are classes that on median it took more than a month.

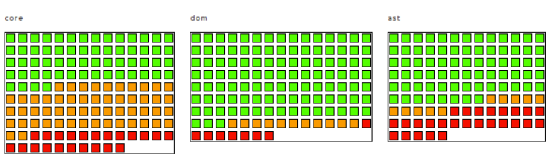

Fig. 10. Bug Lifetime

III. Related Work

Churrasco is a web-based tool for collaborative software evolution analysis [5]. The tool automatically extracts information from a variety of software repositories, including versioning systems and bug management systems. The ultimate goal is to provide an extensible tool that can be used to reason about software evolution under different perspectives, including the behavior of bugs. In contrast, BugMaps has a much stronger historical perspective and offers different metrics. Moreover, BugMaps targets the visual and historical exploration of a single variable (number of bugs). For this purpose, it supports a more rich set of visual measures for reasoning about bugs. Other visualization metaphors have also been provided for understanding the behavior of bugs, including system radiography (which provides a high-level indicator about the parts of the system more impacted by bugs) and bug watch (which relies on a watch metaphor to provide several information about a particular bug) [10]. Hatari [6] is a tool that provides views to browse through the most risky locations and to analyze the risk history of a particular location in a system at the level of lines of code. On the other hand, BugMaps works at the level the of classes and packages.

IV. Conclusions

In this paper we proposed a tool to support retrieval and analysis of bugs stored in bug-tracking systems. The tool extracts time series of defects from such systems and allows the visualization of different bug measures. Its ultimate goal is to facilitate the task of understanding the system with respect to its bugs.

ACKNOWLEDGMENT This research has been supported by grants from FAPEMIG, Brazil and INRIA, France.

References

[1] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *European Software Engineering Conference*, 2005, pp. 1–10.

[2] SonarSource, "Sonar platform." [Online]. Available: sonarsource.org

[3] R. Wettel, "Visual exploration of large-scale evolving software," in *International Conference on Software Engineering*, 2009, pp. 391–394.

[4] M. D'Ambros and M. Lanza, "Bugcrawler: Visualizing evolving software systems," in *European Conference on Software Maintenance and Reengineering*, 2007, pp. 333–334.

[5] M. D'Ambros and M. Lanza, "Distributed and collaborative software evolution analysis with churrasco," *Science of Computer Programming*, vol. 75, no. 4, pp. 276–287, 2010.

[6] J. Sliwerski, T. Zimmermann, and A. Zeller, "Hatari: Raising risk awareness," in *European Software Engineering Conference*, 2005, pp. 107–110.

[7] C. Couto, C. Silva, M. T. Valente, R. Bigonha, and N. Anquetil, "Uncovering causal relationships between software metrics and bugs," in *European Conference on Software Maintenance and Reengineering*, 2012.

[8] T. Girba and S. Ducasse, "Modeling history to analyze software evolution," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, pp. 207–236, 2006.

[9] S. Ducasse, T. Girba, and A. Kuhn, "Distribution Map," in *International Conference on Software Maintenance*, 2006, pp. 203 –212.

[10] M. D'Ambros, M. Lanza, and M. Pinzger, "A bug's life: Visualizing a bug database," in *International Workshop on Visualizing Software for Analysis and Understanding*, 2007, pp. 113–120.