

iProblems - an integrated instrument for reporting design flaws, vulnerabilities and defects

Mihai Codoban[†], Cristina Marinescu*, Radu Marinescu*

[†]“Politehnica” University of Timișoara, Romania

Email: codobanmihai@gmail.com

*LOOSE Research Group, “Politehnica” University of Timișoara, Romania

Email: {cristina.marinescu, radu.marinescu}@cs.upt.ro

Abstract—In the current demonstration we present a new instrument that provides for each existing class in an analyzed system information related to the problems the class reveals. We associate different types of problems to a class: design flaws, vulnerabilities and defects. In order to validate its usefulness, we perform some experiments on a suite of object-oriented systems and some results are briefly presented in the last part of the demo.

I. INTRODUCTION

Many empirical studies answering questions like *Where do bugs come from?* [1] have been done in the last years. Usually those studies rely on two different sources of information: (i) information extracted from the source code (*e.g.*, classes, methods, attributes) which are the building blocks for computing different metrics or metrics-based analyses and (ii) information related to the defects (bugs) extracted from version archives and bug tracking systems that the software entities exhibit. The extracted information is further merged and analyzed most of the times using an automated support for statistical analysis like R. If a researcher or practitioner considers useful to perform a fine-grained inspection of an entity, she has to merge the mentioned sources of information. In this work we present IPROBLEMS, an instrument that allows its users to inspect the design entities of an analyzed object-oriented system in terms of metrics (or metrics-based analyses like design flaws), vulnerabilities and defects.

The general workflow inside IPROBLEMS can be seen in Figure 1. Its inputs are the analyzed system’s source code (and bytecode for vulnerabilities). After parsing the code it creates a model. Using that model it computes different metrics which are combined through detection strategies to detect design flaws on the system’s entities. In the final stage it loads vulnerabilities and defects from different sources, tries to match them to entities (classes) and enriches the model with those vulnerabilities/defects.

II. IPROBLEMS – AN OVERVIEW

In this section we present the different types of information IPROBLEMS can provide us with. Next, for each type of information we dedicate a section.

a) Information related to metrics and metrics-based analyses: in order to compute the values of different metrics (*e.g.*, LOC, DIT) and metrics-based analyses [3] (*e.g.*, Data

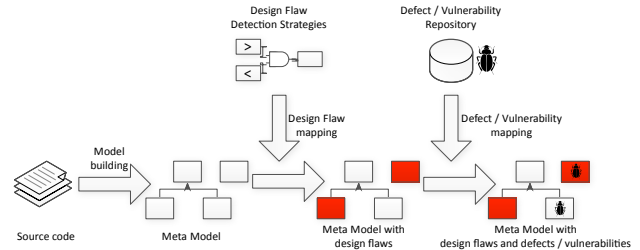


Fig. 1: Overview of the bug mapping process.

Class, Feature Envy) we rely on a model extracted from the source code according to the MEMORIA meta-model [2]. Currently there can be analyzed object-oriented systems implemented in Java, C++ or C#. A detailed list of the provided analyses can be found in [3].

b) Information related to vulnerabilities: currently there are some instruments that allow us to detect possible defects (*e.g.*, vulnerabilities) the classes may exhibit. One well-known tool for detecting vulnerabilities is the static analysis tool FINDBUGS [4]. It runs on bytecode from which it builds several models and checks for bug patterns against those models. Therefore its strength relies on the number and the relevance of the detectors built into it.

There are two types of detectors, *Visitor detectors* which traverse an abstract syntax tree and use a finite state machine to detect patterns in the tree, and *CFG detectors* which use control flow graphs and different data flow analyses to search for patterns (for example, a detector could use a null pointer data flow analysis to propagate the null values across code and then verify if there are null pointer checks put in place).

IPROBLEMS programatically invokes a FINDBUGS analysis over the provided bytecode, gathers the results in the form of *BugInstances* and tries to match each *BugInstance* to an entity from the MEMORIA model. The matching is done via the fully qualified name.

c) Information related to defects: In order to tie defects to source code entities we follow the approach presented in [5], Chapter 27, Mining Your Own Evidence. This approach is based on the following software developers’ good practice that formalizes the way they handle and fix bug reports:

- All defects are reported through a bug tracking system, for example BUGZILLA.

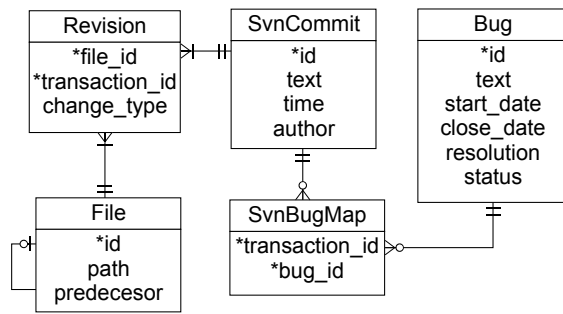


Fig. 2: Defect Mapping Entities.

- Upon committing bug fixing code the developers enter the bug tracking system defect's id (bugID) in the revision's commit message, thus linking a repository transaction to a particular defect. It can occur that some commits involve bug fixes as well as refactorings, but we consider this situations are quite rare.

To tie defects to code entities our tool takes as inputs the xml dumps of the repository and the bug tracking system. While parsing them it forms the database whose structure can be seen in Figure 2. After populating the mentioned database it has to accomplish two tasks: tie bug entries to repository entries and tie software entities to bugs.

- Tying bug entries to repository entries is achieved by searching the commit messages for bugIDs. To avoid spurious mappings we filter the results by special keywords that usually accompany repository commits that fix bugs. By analyzing a number of systems we found out that the following keywords are quite sufficient and covering: fix, fixes, fixed, bugfix, bugfixes, bug, bugs, bugreport, bugzilla, issue, issues.
- Tying software entities to bugs relies on the assumption that every java file contains the definition of a class. We gather all the files that were modified by the revision and try to build a fully qualified name from each file's repository path. If there is a match with any system entity's qualified name we assign the bug to that entity.

One encountered problem is the situation in which a resource is moved into another folder thus breaking its qualified name. Whenever we detect that a file has been moved to another place we link the new name to the old one. Upon building the qualified name for any file we always look back at the first path in the chain. This approach ensures that we always have the first version of the name which is the version that will be linkable back to the code.

III. IPROBLEMS AT WORK

I**PROBLEMS** can be used for assessing object-oriented systems implemented in Java, C++ and C# (the vulnerabilities can be extracted only for Java systems) and we integrate it within the I**PLASMA** [6] front-end called INSIDER in order to be able to use existing quality assessment techniques related

to the source code of an object-oriented system (e.g., metrics, detection strategies [3]).

We inspect different object-oriented systems in evolution, among which *ArgoUML*, *DrJava*, *FindBugs*, *FOP*, *FreeCol* and *JFreeChart*. For example, from the provided information we find out that:

- In *ArgoUml* (Release Date: 2006-02-09) the class that exhibits the most defects reveals many design problems (*ParserDisplay* is a *Brain Class*, contains five methods affected by *Feature Envy* and six *Brain Methods*).
- From *FOP* (Release Date: 2006-04-20) we can notice that the most vulnerable five classes are *Data Classes* (e.g., *InlineContainer*, *PageNumber*, *PageNumberCitation*, *TableAndCaption*, *Character*) and the class exhibiting the most defects is also detected as being affected by design flaws (e.g., *FOPPropertyMapping* contains one *Feature Envy* and one *Brain Method*).
- In *Argouml* (Release Date: 2008-09-26) large interfaces that exhibit many *Shotgun Surgery* design problems (e.g., *Facade*, *TargetManager*) also report defects. Moreover their implementors are usually *God Classes* or *Brain Classes*.
- We found out that there is a correlation between design flaws and defects. On average, 30% of the classes that exhibit design problems also have defects while only 7% of the classes that do not exhibit design problems have defects.

IV. CONCLUSIONS. FUTURE WORK

We present an integrated instrument that provides us with different types of information existing object-oriented entities (classes) may reveal. The provided information can be explored in order to perform different fine-grained analyses at the class level or exported into a csv file and explored in a coarse-grained manner using an adequate tool support for statistical computing and/or data mining.

We intend to integrate our tool into an IDE like Eclipse in order to facilitate its use, as the integration would allow to analyze the system in real-time instead of using a separate tool to construct its model.

REFERENCES

- [1] A. Schröter, T. Zimmermann, R. Premraj, and A. Zeller, "Where do bugs come from?" *SIGSOFT Softw. Eng. Notes*, vol. 31, pp. 1–2, November 2006. [Online]. Available: <http://doi.acm.org/10.1145/1218776.1218791>
- [2] D. Rațiu, *Memoria: A Unified Meta-Model for Java and C++*. Master Thesis, "Politehnica" University of Timișoara, 2004.
- [3] M. Marinescu and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.
- [4] *FindBugs - find bugs in Java programs*, last published: 2009-08-22. [Online]. Available: <http://findbugs.sourceforge.net>
- [5] A. Oram and G. Wilson(editors), *Making Software. What Really Works, and Why We Believe It*. O'Reilly, 2010.
- [6] C. Marinescu, R. Marinescu, P. Mihancea, D. Rațiu, and R. Wetzel, "iPlasma: An integrated platform for quality assessment of object-oriented design." in *Proc. IEEE International Conference on Software Maintenance (ICSM Industrial and Tool Volume)*, Budapest, Hungary. IEEE Computer Society Press, 2005.