# Mining Refactorings in ARGOUML

Peter Weißgerber, Stephan Diehl
University of Trier
Computer Science Department
54286 Trier, Germany
weissger@uni-trier.de, diehl@acm.org

Carsten Görg*
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
goerg@cc.gatech.edu

## ABSTRACT

In this paper we combine the results of our refactoring reconstruction technique with bug, mail and release information to perform process and bug analyses of the ARGOUML CVS archive.

**Categories and Subject Descriptors:** D.2.8[Software Engineering]:Metrics; D.2.5[Software Engineering]:Testing and Debugging

**General Terms:** Algorithms, Management, Measurement.

**Keywords:** Refactoring, mails, bugs, evolution, re-engineering.

## 1. INTRODUCTION

In this study we mine the CVS archive of ARGOUML for refactorings that have been performed during the development and evolution of ARGOUML. We relate the refactorings of each day to the number of overall changes on that day to detect both phases with many and phases with almost no refactorings. We look especially at the phases before major release dates, because this may help the project manager in planning pre-release phases, or to plan release dates ahead.

To see if refactorings in ARGOUML have an effect on the occurrence of new bugs and on communication between the developers, we relate the refactorings to bug reports in ISSUEZILLA respectively to mails on the developer mailing list. If the error rate would increase with the refactoring ratio, the project manager would have to enforce the use of automated refactoring tools, or the used refactoring tools or methods may be poor.

Finally, we examine if there are incomplete refactorings which could possibly lead to errors. Mining changes for such incomplete refactorings can uncover bugs that have been introduced long ago.

## 2. MINING REFACTORINGS IN ARGOUML

### 2.1 Computing the Refactoring Ratio

In [1] we introduced our technique to reconstruct refactorings from software archives such as CVS. For each day, we determine which blocks (fi elds, methods in a class) have been changed and which of these are affected by refactorings. Thus we get values for the following metrics:

**Normalized number of changed blocks (per day):**
$\%CB_d = \frac{\#CB_d}{\#CB_{max}}$
where $\#CB_{max} = max\{\#CB_d | d \text{ day in project's lifetime}\}$

**Number of refactorings per changed block (per day):**
$\%RB_d = \frac{\#RC_d}{\#CB_d}$ where $\#RC_d$ is the number of non-overlapping, disambiguated refactoring candidates for day $d$

### 2.2 Computing Bug and Mail Ratios

To determine if days with a high refactoring ratio result in fewer errors than other days, we look at the number of bugs fi led per day in the ISSUEZILLA system of ARGOUML. As developers usually do not detect errors immediately after the program change that caused them, we compute the number $OB_d$ of all new *defects filed within the next five days* (which roughly approximates a working week). For each day we relate this value to the number of changes:

**Normalized number of bugs per changed block:**
$\%BB_d = \frac{\#OB_d}{\#CB_d \#OB_{max}}$
where $\#OB_{max} = max\{\#OB_d | d \text{ day in project's lifetime}\}$

Additionally, we are interested in whether refactorings have an effect on the amount of communication between the developers. Therefore, we consider the development mailing list of ARGOUML and count for each day the number $AM_d$ of archived mails. We relate this number to the number of changes as follows:

**Normalized number of mails per changed block:**
$\%MB_d = \frac{\#AM_d}{\#CB_d \#AM_{max}}$
where $\#AM_{max} = max\{\#AM_d | d \text{ day in project's lifetime}\}$

## 3. ARGOUML RESULTS

### 3.1 Process Analysis: The Pre-Release Phase

Figure 1 shows the time periods before and after the four release dates of the stable series of ARGOUML from 2002 until 2005. In all cases, we see the same pattern:

- Before the release dates, there seem to be testing phases where only few changes have been done at all, but many new bug reports have been opened.
- In each case after the testing phase and immediately before the release, changes with high refactoring ratio have been performed.

*The author was supported by a fellowship within the Postdoc-Program of the German Academic Exchange Service (DAAD).


*MSR'06*, May 22–23, 2006, Shanghai, China.

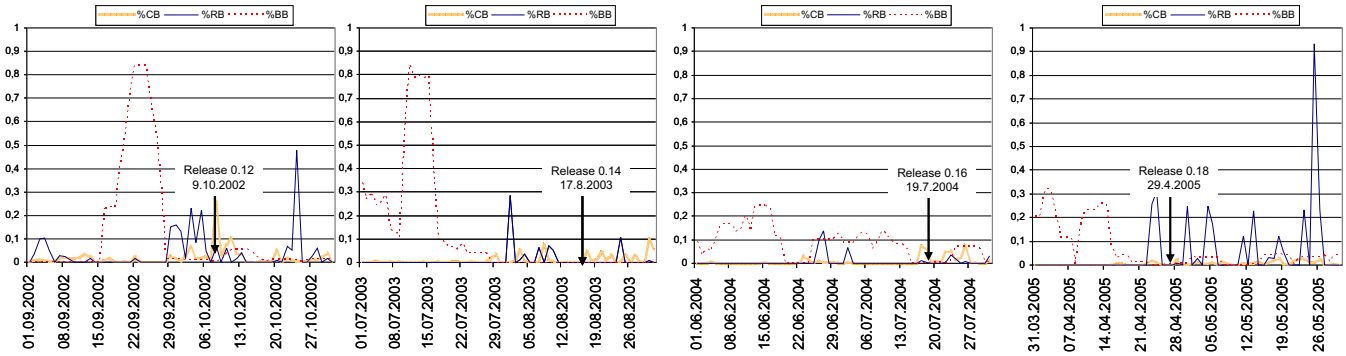


**Figure 1: Relative number of changes, refactorings and bugs before major releases.**

## 3.2 Bug Analysis: Correlation between Refactorings, Mails, and Bugs

Figure 2 shows the values of the normalized number of bugs per day $\%BB_d$, as well as the normalized number of mails per day $\%MB_d$ compared to the refactoring ratio per day. While the Spearman correlation between $\%RB$ and $\%BB$ is only about 0.3, it stands out that after days with a high refactoring ratio only few bug reports have been opened in the bug tracking system. The same holds for mails: When the refactoring ratio is high, few mails have been written. However, we are aware that these correlations could be accidental or caused by other factors like feature freezes that we did not yet take into account.

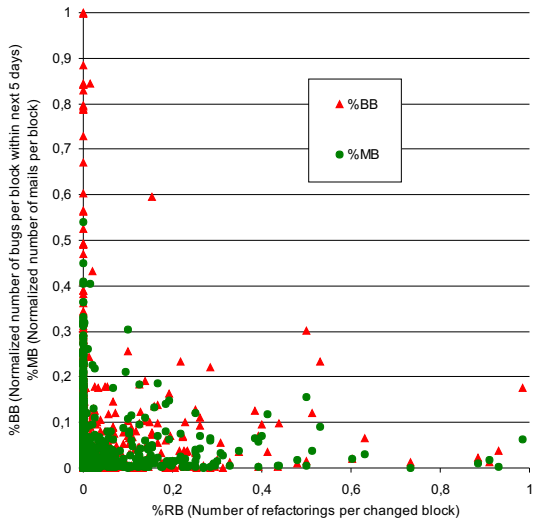


**Figure 2: Few bug reports and mails after days with high refactoring ratio.**

## 3.3 Bug Analysis: Incomplete Refactorings

Refactoring reconstruction can also be used to detect incomplete, and thus erroneous refactorings [2]. In these cases, parameters have been added to or removed from methods, but the developer did not change the corresponding methods in super-, sub-, or sibling classes accordingly. In ARGOUML we found 33 transactions containing such incomplete refactoring candidates between Jan 2003 and Dec 2005.

Figure 3 shows a candidate for a possibly incomplete refactoring: the sibling classes `ActionSaveGraphics` and `ActionSaveProjectAs` both contained the method `trySave(bool)`. The refactoring *RemoveParamter* was applied only to the method in the class `ActionSaveGraphics` and possibly it also should be applied to the method in the class `ActionSaveProjectAs`.

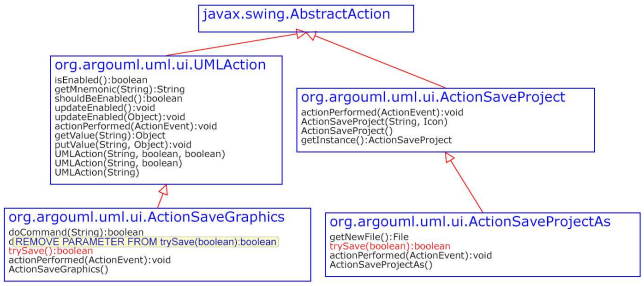


**Figure 3: Missing Remove Parameter refactoring.**

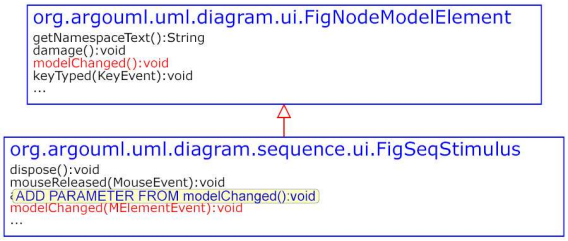


**Figure 4: Missing Add Parameter refactoring.**

Figure 4 shows the application of an *AddParameter* refactoring to the method `modelChanged()` in the class `FigSeqStimulus`. The refactoring has not been applied to the method `modelChanged()` in its superclass `FigNodeModelElement`. However, some transactions later the *AddParameter* refactoring has been applied to the method `modelChanged()` in the superclass and also to methods in five other subclasses of `FigNodeModelElement`. Apparently the refactoring was incomplete in the beginning.

**Acknowledgments.** Michael Stockman kindly provided the bug data for ARGOUML.

# 4. REFERENCES

[1] C. Görg and P. Weißgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of International Workshop on Program Comprehension (IWPC05)*, St. Louis, Missouri, USA, May 2005.

[2] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *Proceedings of International Workshop on Mining Software Repositories (MSR05)*, St. Louis, Missouri, USA, May 2005.