Are Refactorings Less Error-prone Than Other Changes? *

Peter Weißgerber
University of Trier
Computer Science Department
54286 Trier, Germany
weissger@uni-trier.de

Stephan Diehl
University of Trier
Computer Science Department
54286 Trier, Germany
diehl@acm.org

ABSTRACT

Refactorings are program transformations which should preserve the program behavior. Consequently, we expect that during phases when there are mostly refactorings in the change history of a system, only few new bugs are introduced. For our case study we analyzed the version histories of several open source systems and reconstructed the refactorings performed. Furthermore, we obtained bug reports from various sources depending on the system. Based on this data we identify phases when the above hypothesis holds and those when it doesn't.

Categories and Subject Descriptors

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

General Terms

Algorithms, Management, Measurement

Keywords

Refactoring, software evolution, reverse engineering, and re-engineering

1. INTRODUCTION

Changes to source code can be roughly categorized as bug fixes, feature extensions, and refactorings. Intuitively, we expect that feature extensions are more error-prone than bug fixes or refactorings. By definition, a refactoring should not alter the program behavior at all and, thus, not introduce any new bugs. Thus, during phases in the program development where refactorings prevail, we would expect that less errors are introduced than in other phases. In this paper we present the results of a first case study which relates the percentage of refactorings per day to the ratio of bugs opened within the next 5 days.

The remainder of this paper is organized as follows: In Section 2 we describe how we reconstruct refactorings and other changes from version archives. Then, we explain in Section 3 how we relate these changes to bugs that have emerged in the lifetime of a software system. For the case study in Section 4 we applied our technique to three open-source systems. Section 5 discusses related work and Section 6 concludes this paper.

2. OBTAINING REFACTORINGS

In this section we briefly explain our technique to extract refactorings from software version archives such as CVS. The details of this technique can be found in [5].

A prerequisite for detecting refactorings that occurred during the evolution of a software system is to collect information about all changes that have been stored in the archive. After that, these change can be analyzed to decide if they are a refactoring, part of a refactoring or no refactoring at all.

As we focus in this paper on the question of whether refactorings are less error-prone than other changes, we have to compute the ratio of refactorings to other changes. In particular, this allows us to look for those days (or weeks, months, ...) which had a high refactoring ratio.

2.1 Recovering Basic Change Information

To recover the changes performed to a software system during its evolution we analyze the software archive as presented in [9]. As a result we obtain the following information:

- **Versions**: A version describes one revision of one file in the software archive, for example the revision 1.2 of the file `src/Main.java`. For each version we extract the following information: the filename and the revision number, the revision number of the predecessor version (i.e. the version of the same file that has been changed to create this particular version), the developer who checked-in the version into the archive, the log message and the timestamp of the check-in, the state of the version (e.g., "dead" for versions that are not used anymore), the numbers of added, altered, and deleted lines, and the version text.
- **Transactions:** A transaction is defined as the set of versions that have been checked-in into the version archive by an author in one commit operation. As

*(Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.

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

CVS splits each commit operation containing multiple versions into separate check-ins for each version, we use a sliding time window heuristic to recover transactions quite precisely. For every transaction recovered this way, we additionally store the timestamp of the start, the length, as well as the committer and the log message of the transaction.

For our study we use the transactions as a heuristics which changes have been performed at the same time and, thus, might be related to each other. For our purpose, we are not interested in the numbers of the lines that have been changed, but in the names of the affected code-blocks — these are classes, fields and methods. Figure 1 gives an overview of the overall process of computing these numbers which is described in the following sections.

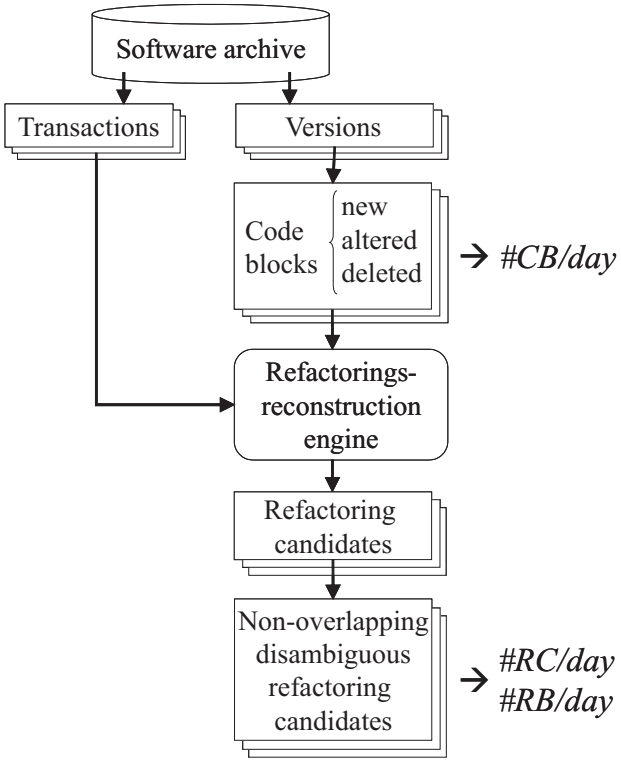

Figure 1: Computation of Changed Blocks and Refactorings

2.2 Finding Changed Code Blocks

To determine which code blocks have been changed in a version with respect to its predecessor, we first have to compute the blocks which are actually contained in it. For this we compute for each version v the following sets:

- the set C_v of classes / interfaces (identified by their fully-qualified class name),
- the set F_v of fields (identified by their signature, consisting of name and type),
- and the set M_v of methods (identified by their signature consisting of name, parameters, and type).

For each block we additionally store the start line and the end line. This allows us to reconstruct the nesting of the symbols in a file, and to distinguish methods that have the same signature but belong to different classes defined in the same file.

Let $B_v = C_v \cup F_v \cup M_v$ the set of blocks contained in version v.

Next, for each version v we compare its set of code blocks with the set of code blocks for the predecessor version v' (for versions that have no predecessor, we take the empty version as v', thus we compare with the empty set). Thus we compute the following sets:

- $B_v - B_{v'}$: new code blocks, i.e. blocks that only exist in the newer version
- $B_v \cap B_{v'}$: common code blocks, i.e. blocks that exist in both versions
- $B_{v'} - B_v$: deleted code blocks, i.e. blocks that only exist in the predecessor version

Obviously, every new code block and every deleted code block is affected by the transition from v' to v. Moreover, common code blocks may also have been altered and, thus, affected by the transition from v' to v.

In our current implementation we use a light-weight regular-expression based parser to compute the code blocks for JAVA files, and perform a textual comparison to find out if a code block has been altered in the transition from version v' to its subsequent version v. Another possibility to get this information is to use the ECLIPSE [8] structural compare plug-in which is able to identify and compare blocks for several file types and programming languages.

2.3 Refactorings

In the previous section we have explained how we extract information about changed JAVA code blocks (classes, methods, field). As we have also recovered transactions we now know which blocks have been affected by changes performed at the same time. Next, we want to determine which of these changes are refactorings, and thus, which of the changed blocks are affected by refactorings.

Our refactoring reconstruction engine [5] takes the information about changed blocks and transactions as input and yields for each transaction a set of refactorings. We call these refactorings *refactorings candidates*, to indicate that they have possibly been performed in this transaction. In this paper, we address refactoring kinds for which candidates can be computed by comparing the signatures and contents of added and removed code blocks. Other refactoring kinds require more semantic information like type inferences or the class hierarchy. For this study we compute refactoring candidates of the following kinds:

- Move/Rename class/interface $c_1 \Longrightarrow c_2$, where c_1 is the old fully-qualified class name and c_2 the new one.
- Move field $(f, c_1) \Longrightarrow (f, c_2)$, where f is the field signature, c_1 is the fully-qualified name of the old class of the field, and c_2 is the fully-qualified name of its new class.
- Move method $(m, c_1) \Longrightarrow (m, c_2)$, where m is the method signature, c_1 is the fully-qualified name of the

old class of the method, and c_2 is the fully-qualified name of its new class.

- Rename method $m_1 \Longrightarrow m_2$, where m_1 is the old signature of the method and m_2 the new one.
- Hide/Unhide method m, where m is the signature of the method that has been hidden respectively unhidden.
- Add/Remove parameter to/from method $m_1 \Rightarrow m_2$, where m_1 is the old method signature and m_2 the new method signature.

However, in this paper we are mainly interested in how many refactorings are performed, but not in the kind of each refactoring. To count the number of refactorings the following issues must be considered:

Overlapping refactorings Our refactoring reconstruction engine both detects refactoring candidates on class level (move/rename class/interface) as well as on a fine-grained level (refactorings on fields and methods). The problem with this is, that refactorings on class level automatically *include* refactorings on the fine-grained level: If a class is moved or renamed, automatically all fields and methods in it are moved to the new location and, thus, are counted. So if we counted both refactorings candidates on class level and on the fine-grained level, we would count some refactorings twice. To solve this problem, we omit the class-level refactoring candidates in our count.

Ambiguous refactorings As the refactoring reconstruction engine only yields refactoring *candidates* (because it cannot decide whether a program transformation really preserves the program behavior or not), for the same transaction several refactoring candidates may be suggested that are ambiguous and may exclude each other. For example, assume that the reconstruction would suggest the following refactoring candidates:

1. Move Method $(m, c_1) \Rightarrow (m, c_3)$.
2. Move Method $(m, c_1) \Rightarrow (m, c_4)$.
3. Move Method $(m, c_1) \Rightarrow (m, c_5)$.
4. Move Method $(m, c_2) \Rightarrow (m, c_3)$.

In this example the first and fourth candidate are ambiguous concerning the source. If both are taken into account, this would mean that two different methods (the one with signature m in class c_1 and the one with the same signature in c_3) are moved to the same target. Although such operations are possible (the methods may have been identical) it is likely that at most one of both candidates is correct. A similar problem occurs with the first three candidates: There are three alternatives to which class the method has been actually moved.

Thus, instead of counting the number of all refactoring candidates, we compute the number of unambiguous refactoring candidates. A conservative approximation for this number is to take as many refactoring candidates into account as there are different sources respectively targets, depending on which number is smaller. In the example above there are two different sources, namely (m, c_1) and (m, c_2), and three different targets: (m, c_3), (m, c_4), and (m, c_5). Thus the number of unambiguous refactoring candidates for this example is 2.

Number of affected blocks The number of affected blocks of a refactoring only depends on the kind of refactoring: Refactorings of kinds move field, move method, rename method, add parameter, and remove parameter affect two blocks because they change the signature of the refactored artifact and, thus, create a new block. In contrast, refactorings of kind hide/unhide method do not change the signature of the refactored block and, therefor, affect only one block.

3. RELATING REFACTORINGS TO BUGS

As the goal of this study is to see whether the common belief, that refactorings are less risky than other changes, is really true, we try to estimate how many issues have emerged because of a change. To this end, we first describe how we get information about the bugs that appeared in a software system during its lifetime. Then we describe how we relate changes and refactorings to these bugs.

3.1 Obtaining Bug Information

Depending on the project, information can be obtained from the following sources:

Bug databases: The most obvious way to gather bug information is to directly access the bug database. Unfortunately, only project administrators have direct access to the database, the average developer and other people have to use a web interface to query and alter the bug database. Querying the database for every existing issue over the web interface can be tedious if we want to retrieve all information available for all bugs. However, getting an overview on the bugs is easy using the web interface. This overview contains for every bug at least its ID, and a hyperlink to additional information about the bug, that can be downloaded and parsed if needed. For SOURCEFORGE projects the overview also includes the summary of each bug, the date when the bug has been opened, the priority, the status of the bug, the developer it is assigned to, and the developer who has submitted it. In contrast, the BUGZILLA overview does not contain the bug open date and the submitter, but instead the severity of the bug, the affected computer platforms, and for closed bugs also the resolution (which may be "bugs is fixed" or "bug report was invalid").

Bug report mails: Bug report mails are emails that are automatically generated and sent to the developer mailing list by bug databases such as BUGZILLA or SOURCEFORGE when a new bug has been opened in the database or the entry of an existing bug has been changed. A bug report mail at least contains information about the ID of the bug that has been altered, and a textual description what exactly has been changed (e.g., the bug has been resolved, or a comment has been added).

To get bug information using bug mails, one needs to have access to the email archive of the project. Fortunately, for most projects—especially those hosted at

SOURCEFORGE—these archives are freely accessible over the web.

Recognizing the bug report mails in the mail archives is quite simple: the sender of this mail is the bug database and normally, the subject has a special format. For example, the SOURCEFORGE bug database sends all its mails with the sender `noreply@sourceforge.net` and the subject "`[ project-Bugs-ID ] bug description`".

Next, we parse the bug report mails using a regular-expression based parser. In this work we are mainly interested in when a bug has been opened. But other information can also be recovered using appropriate regular expressions.

3.2 Relating Changes and Bugs

Unfortunately, although bug information can be retrieved from bug databases and bug report mails as described above, there are only very few cases where the bug information contains exact specifications about which source code change is responsible for the respective bug. Thus, we have to use heuristics here as well.

Our heuristics is to assume that a bug may be caused by changes that have been done in a time window of n days before the bug has been opened (reported).

Thus, for every day d in the lifetime of a project, we compute the number of bugs $\#OB_d^n$ that have been opened at day d and the next n days.

Furthermore, we have to take into account that there are days when there is more activity and when there is less. As a measure for the activity we use the number of blocks changed per day.

Thus, for every day d the number of changed blocks is $\#CB_d = \sum_{t \in T_d} \#CB_t$ where T_d is the set of transactions on day d.

Instead of looking at the absolute numbers of refactorings and bugs per day, we relate them to the number of changed blocks. We also relate $\#OB_d^n$ to the number of changed blocks to be able to compare it with the refactoring ratio.

As the refactoring ratio is a percentage, its values are in the range of [0, 1]. To be able to draw a single diagram containing the refactoring ratio, as well as the number of changed blocks and the number of bugs per block, we normalize the latter two by dividing by the maximum value.

Thus, in our study we compared the following three metrics:

Normalized number of changed blocks:
$\%CB_d = \frac{\#CB_d}{\#CB_{max}}$ where
$\#CB_{max} = max\{\#CB_d | d \text{ day in the projects lifetime}\}$.

Normalized number of bugs per changed block:
$\%BB_d^n = \frac{\#OB_d^n}{\#CB_d \# OB_{max}^n}$ where
$\#OB_{max}^n = max\{\#OB_d^n | d \text{ day in the projects lifetime}\}$.

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

Project	in CVS since	#txns	#versions	#dev
ARGOUML	1998-01-26	16138	65593	42
JEDIT	2001-09-02	2141	10726	6
JUNIT	2000-12-03	832	1707	6

Figure 2: CVS data for the analyzed project txns = transactions, dev = committers

4. CASE STUDY

4.1 The Analyzed Projects

In our case study we have applied the described techniques to three different open source projects: JEDIT, JUNIT, and ARGOUML. Although these projects may not be representative for all open-source projects, they are very different in age, size, number of transactions, and number of involved committers, as Figure 2 illustrates.

4.2 Study Settings

We applied our technique described in the previous sections to these three projects as follows: First, we computed for each transaction the number of total changed code blocks and the number of blocks affected by refactorings. To get information on the level of days, we computed the sum of the number of changed code blocks as well as of the number of code blocks affected by refactoring for each transaction that has been started at the same day.

Next, we collected data about how many bugs have been opened per day for these projects. For ARGOUML a developer created bug statistics [7] and kindly provided the raw data of these to us. For JEDIT we relied on the bug report mails that are sent by the project's bug database to the developer mailing list. As JUNIT does not use such bug report mails we extracted the bug information for this project from the SOURCEFORGE web interface.

As explained, we computed the value $\%BB_d^n$ that relates the changes of day d to bugs opened within the next n days. Obviously, looking only at the same day would not be sufficient. Thus, we decided to use a longer time window of $n = 5$ which roughly corresponds to a working week.

4.3 ARGOUML

For ARGOUML Figure 3 shows for each day the normalized number of changed blocks $\%CB$, the percentage of refactored blocks per day $\%RB$, and the normalized number of bugs per changed block $\%BB$.

It can easily been seen that for most days the percentage of refactorings with respect to all changed blocks is rather small, interestingly there is no day where all changed blocks are affected by refactorings. The days with the highest refactoring percentages are mainly between April 2005 and October 2005, thus we look at this period in more detail (see Figure 5).

When we look at this figure it seems that for most days with a high refactoring rate, the value of $\%BB$ does not change tremendously. But, interestingly, for June 30, the day with the overall highest refactoring rate (98.4%), it increases noticeably. The log messages for the two transactions performed on that day state that the undo functionality is moved to a package on its own, but that a new word-wrap feature is introduced additionally.

Another day with a noticeable refactoring rate and increasing $\%BB$ is July 31. On the transactions at this day,

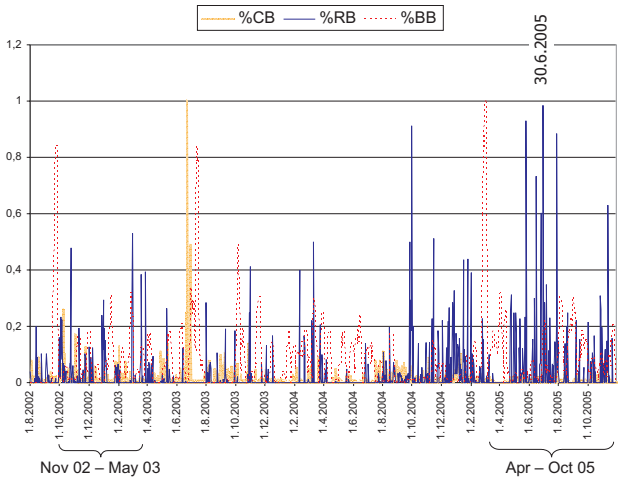

Figure 3: Overview of values computed for ARGOUML

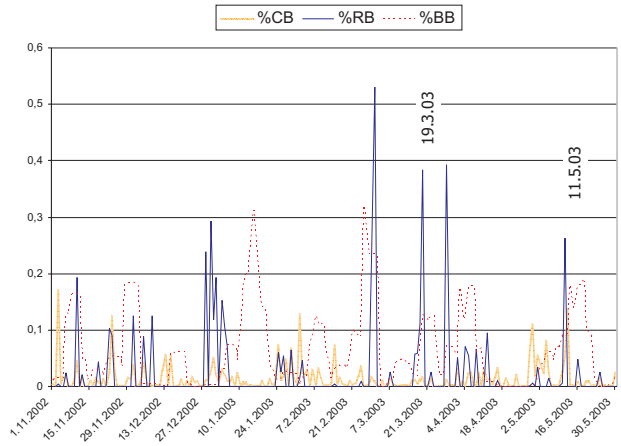

Figure 4: ARGOUML: November 2002 to May 2003

according to the log messages, the class `PropPanelSignal` has been refactored, and furthermore among other things, the developers have "implement(ed) signals and timexpressions for events". When looking at the bugs that have been filed within the next five days we found a bug with the summary "Attributes disappear after typing an initial value" in the component `PropertyPanel`, which also contains the refactored class.

Another phase that contains many refactorings (although the ratio is not very high for these days) is between November 2002 and May 2003—this phase is shown in Figure 4. For most days with a high refactoring rate we cannot see an effect on the normalized number of bugs per changed block, with two exceptions: On March 19 and on May 11 we detected a high refactoring rate (39% resp. 26% of the changed blocks affected by refactorings) but the value of $\%BB$ increases as well. We looked in detail at these two dates: the most noticeable specific characteristic of these days is that they contain quite many transactions: On average between 5 and 6 transactions have been performed per day, but on these days the transaction count has been 16 respectively 21.

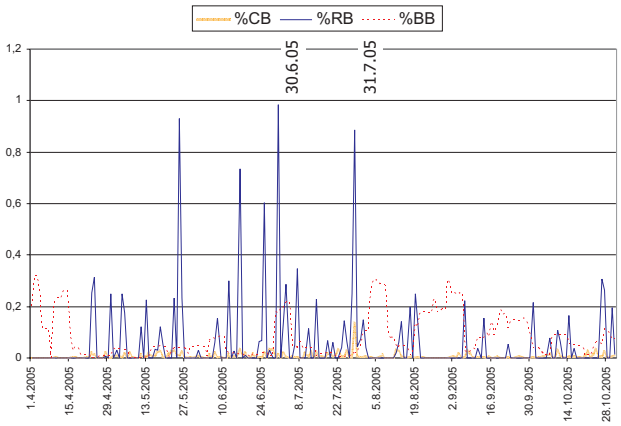

Figure 5: ARGOUML: April to October 2003

4.4 JEDIT

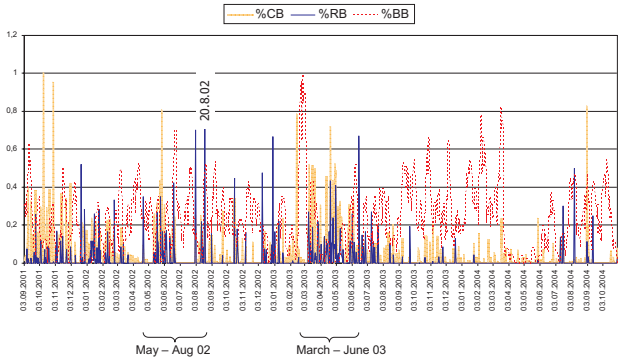

Figure 6: Overview of values computed for JEDIT

Figure 6 illustrates our results for JEDIT. Again it catches our eyes that there is no day that contains only refactorings covered by the kinds we recognize. The highest refactoring rate is even lower than $\frac{3}{4}$: it is 73.4% at August 20 2002.

We zoomed into two periods that seemed to be interesting because there are many refactorings: Figure 7 gives a closer look at the period between May and August 2002, while Figure 8 shows the period between March and June 2003. The following paragraphs describe these two phases in more detail.

4.4.1 JEDIT refactoring phase in 2003

Let us first look at the refactoring phase in 2003 (Figure 8): It attracts attention that between April 19 and May 3 a lot of changes with a high percentage of refactorings have been done, but no, respectively very few, new bugs have been introduced in these days. This seems to support the thesis that refactorings are changes that are not error-prone. We inspected the log messages given for the concerned transactions and found out that the developers documented mainly fixes and refactorings, but only few new features in this phase.

There are also other days between March and June 2003 when changes with a high percentage of refactorings have been done and no—or even a decreasing—effect on the value

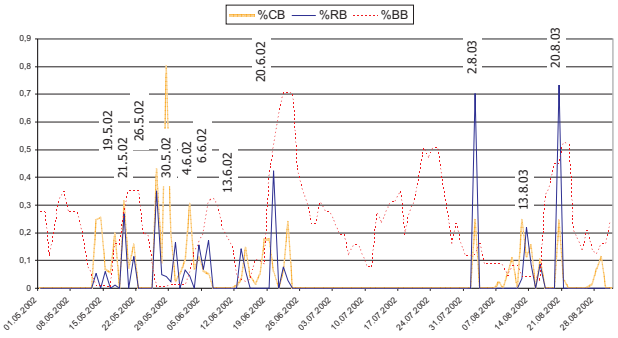

Figure 7: JEDIT: May to August 2002

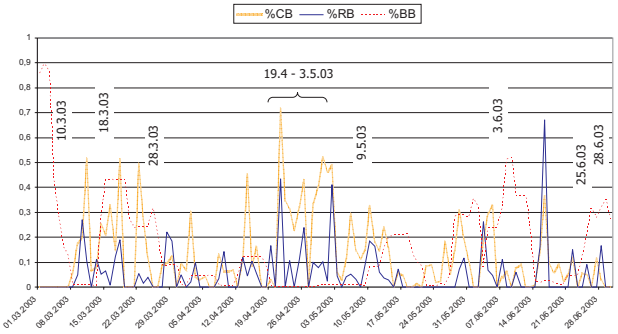

Figure 8: JEDIT: March to June 2003

of $\%BB$ can be observed: The respective days (March 3, March 18, March 28) are annotated in Figure 8.

But we also found days where the refactoring percentage $\%RB$ is rather high but the normalized number of bugs per changed block increases nevertheless. This holds, for example, for May 9, June 7 until June 9, and June 25 until June 28. We manually looked at the log messages of the transactions performed on these days, as well as on the source code changes. Although the refactoring percentage is high for these days, functionality changes have been performed, even within the blocks affected by refactorings.

4.4.2 JEDIT refactoring phase in 2002

Figure 7 shows the refactoring phase from May 2002 until the end of August 2002. There are some days (May 21, May 26, May 30, June 13, Aug 8, Aug 13) with a high refactoring ratio that, as expected, do not cause the normalized number of bugs per changed block to increase noticeably.

But for four days (May 19, Apr 6, May 20, Aug 20) with a refactoring percentage of greater than 10%, the value of $\%BB$ increases. At the first of these dates, more than a quarter of the changed blocks have been affected by refactorings. Two days later again 10% of the changed blocks have been affected by refactorings. Nevertheless, the normalized number of bugs per changed block has a peak at these days. We looked in the log messages of the respective transactions to find evidence for the author's intention of the changes and found that there has been a "display code rewrite", "syntax and text area reworking" and several "TokenMarker code refactorings" and "syntax refactoring" (all transactions have been performed by the same developer). Interestingly, one of the bugs that has been filed within 5 days has the summary "text area and syntax packages: Major redraw issues". However, the developer who committed the changes was expecting such problems: He has commented the bug: "The code in CVS is a work in progress. It might not even compile [...] it might not even run. [...] I'm [...] rewriting [...] the syntax highlighting code; so problems [...] are to be expected."

4.5 JUNIT

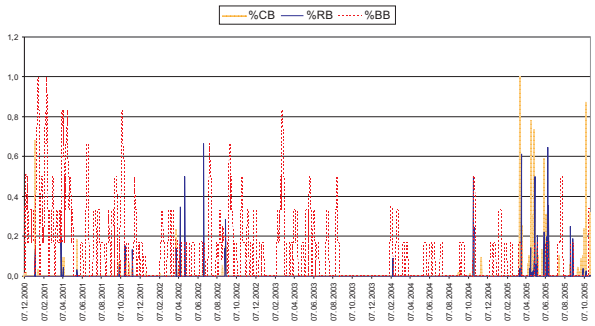

Figure 9: Overview of values computed for JUNIT

We repeated our experiments for JUNIT, the complete results are illustrated in Figure 9. Like for the other projects, there is no day with a refactoring percentage of 100%. The top refactoring percentage for JUNIT is even smaller than for the other two analyzed projects: on June 25 2002 exactly $\frac{2}{3}$ of all changed blocks have been affected by refactorings.

Although we found only refactorings at a few days, there seem to be two phases when refactorings have taken place: The first one between March and September 2002, and the second one between March and October 2005.

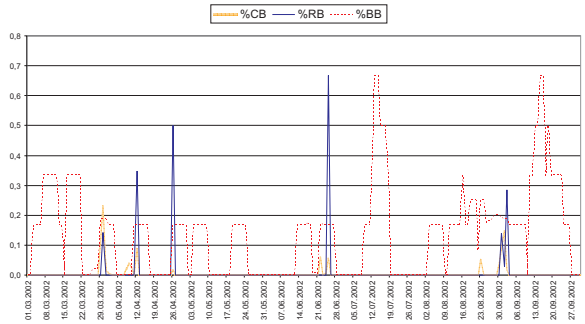

Figure 10: JUNIT: March to September 2002

Figures 10 and 11 show these two phases in more detail. It seems that in 2002 days with a high refactoring rate are likely to be followed by new bug reports, while in 2005 refactorings are rather done after bug reports have been filed.

5. RELATED WORK

Several techniques for detecting refactorings that occurred between subsequent versions of a software system have been developed [2, 4, 1, 5].

In a previous paper we showed that extracted refactoring candidates can be checked for completeness, i.e., whether all

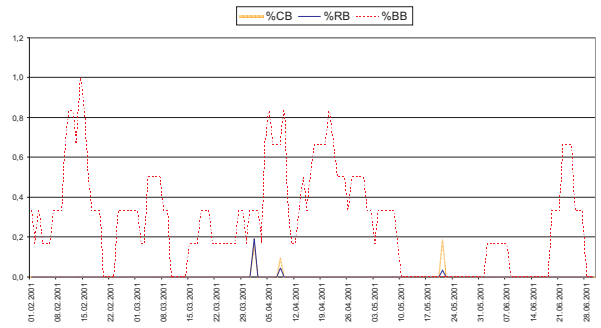

Figure 11: JUNIT: March to October 2005

related locations have been changed [6].

In a recent case study Dig and Johnson found that 80% of API changes, that lead to errors in the applications using these APIs, are refactorings [3].

6. CONCLUSIONS

In this paper we have presented a technique to relate refactoring candidates that we extracted from CVS archives for JAVA programs to bug data in order to find out if the ratio of refactorings with respect to all changes has an impact on the number of bugs that arise in the next days. Although we found interesting correlations between refactorings and bug reports, we are aware that these could be accidental or caused by other factors like feature freezes that we did not yet take into account.

In our case study we applied this technique to three open-source projects. It turned out that in all three projects, there are no days which only contain refactorings. This is quite surprising, as we would expect that at least in small projects like JUNIT there are phases in a project when only refactorings have been done to enhance the program structure. But actually by far the highest refactoring ratio occurred in ARGOUML which is by far the largest one of the projects.

Finally, we found phases of the projects where a high ratio of refactorings was followed by an increasing ratio of bugs, as well as phases where there was no increase. While phases of the second type prevail, phases of the first kind give interesting insight when and why refactorings can cause errors.

7. ACKNOWLEDGMENTS

Michael Stockman kindly provided the bug data for ARGOUML.

8. REFERENCES

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *Proceedings of 7th International Workshop on Principles of Software Evolution (IWPSE 2004), 6-7 September, Kyoto, Japan*, pages 31–40. IEEE Computer Society, 2004.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000)*, pages 166–177, Minneapolis, Minnesota, USA, 2000. ACM Press.

[3] D. Dig and R. Johnson. The role of refactorings in API evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 389–398, Budapest, Hungary, 2005. IEEE Computer Society.

[4] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[5] 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.

[6] C. Görg and P. Weißgerber. Error Detection by Refactoring Reconstruction. In *Proceedings of International Workshop on Mining Software Repositories MSR 2005*, St. Louis, Missouri, USA, May 2005.

[7] M. Stockman. ARGOUML statistics and diagrams homepage. `http://user.tninet.se/~zaa397e/argouml/`.

[8] The Eclipse Foundation. Eclipse Homepage. `http://www.eclipse.org`.

[9] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR04)*, Edinburgh, Scotland, UK, May 2004.