Mining a Change-Based Software Repository

Romain Robbes
Faculty of Informatics
University of Lugano, Switzerland

Abstract

Although state-of-the-art software repositories based on versioning system information are useful to assess the evolution of a software system, the information they contain is limited in several ways. Versioning systems such as CVS or SubVersion store only snapshots of text files, leading to a loss of information: The exact sequence of changes between two versions is hard to recover. In this paper we present an alternative information repository which stores incremental changes *to the system under study, retrieved from the IDE used to build the software. We then use this change-based model of system evolution to assess when refactorings happen in two case studies, and compare our findings with refactoring detection approaches on classical versioning system repositories.*

1 Introduction

The nature of information found in software repositories determines what we can infer from it. Conversely, information missing from a software repository hampers the quality of the research we perform: What is stored in a repository is of prime importance. However, another characteristic limits the choice of source code repositories: the number of available case studies[12]. This pragmatic reason explains why researchers base themselves on popular repositories such as CVS and SubVersion despite their limitations. Indeed, most open source projects give free access to their repositories, including industrial-size case studies such as Apache, Eclipse, Mozilla or Linux.

This large availability comes with a price. Versioning systems are designed to be used in a variety of contexts and hence must lower their assumptions about the objects they version. CVS and SubVersion – the most popular versioning systems in the open-source world – only assume that the objects they version are files. They are thus used in a variety of situations, from source code files to system documentation or binary files. Even if Estublier *et al.* write in [6], that one of the next steps for versioning systems research is to break the assumption of language independence, this is not yet the case in practice. We claim that these assumptions are too weak for researchers to perform precise research on source code evolution. Basing an analysis only on the successive versions of a source tree of code files implies a heavy pre-processing to raise the abstraction level beyond files and directories. Versioning systems have another limitation degrading the quality of the information they contain: They only update their repositories when a developer explicitly checks in his work. Ideally, updates to the repository should come often to be as small and precise as possible, but this cannot be guaranteed.

This paper explores the benefits and drawbacks obtained by breaking the assumption that a popular repository must be used. We instead created a software repository designed to store a maximal amount of information about an evolving piece of software. In particular, we do not use a versioning system, but built from the ground up a *change-based* software repository which fetches domain-specific information from an Integrated Development Environment (IDE). Being change-based means that the evolution of a software system is not modelled as a sequence of versions anymore: Using an IDE allows us to store, as first-class citizens, the actual changes which were performed on the system to obtain its latest version. This model better matches the actual evolution of a system since we reproduce how developers actually change the system.

To validate our approach we chose to study when a particular kind of changes, namely refactorings, were applied to a software system, based on two case studies. We compare the findings of this application of our approach with other work in which refactorings are detected in a classical versioning system repository.

Structure of the paper. Section 2 details the limitations of versioning systems as evolutionary data repositories. Section 3 shows how the change-based repository we implemented addresses these concerns. Section 4 shows a concrete application of our approach: An analysis of the frequency of refactoring operations in two case studies. Section 5 discusses our approach while Section 6 concludes the paper and outlines future work.

Fourth International Workshop on Mining Software Repositories (MSR'07)
0-7695-2950-X/07 $20.00 © 2007 IEEE

2 Limitations of Versioning Systems

Versioning systems, especially those chosen for their popularity among developers – such as CVS and SubVersion –, have two shortcomings which limit the amount of information we can recover from them: They are file-based and snapshot-based [12] [2].

A file-based versioning system versions files and lines. Since all documents in a computer are represented as files, a file-based versioning system can version anything. This lowest-common denominator approach has allowed file-based versioning systems to become ubiquitous, relegating domain-specific versioning systems to niche markets.

However, text files are a poor medium to perform precise source code analysis. A heavy pre-processing must be performed to raise the abstraction level beyond file and lines: parsing every version of the system to build a model of it, and then linking these models together to obtain the overall history of the system in terms of versions. This process needs parsers and dedicated tools, and is costly in terms of implementation and computing time.

Snapshot-based versioning systems store updates in their repositories as deltas between two snapshots of the system. They do so when a developer commits to the versioning system. The issue is that the larger the delta between two versions is, the harder it is to distinguish individual changes. Since versioning systems only rely on notifications from the developer, they cannot guarantee small deltas. An exception is the approach presented by Schneider *et al.* in [15], in which changes are automatically committed to a secondary repository. However the interval at which this is done is not specified, and the source code might not be in a compilable state.

Most developers submit their changes only when a task is finished or at the end of their working day. Hours or days can go by during which the versioning system is not notified of changes. When a commit is finally issued, it will have aggregated all those changes in a single transaction. Changes tend to get muddled together: It becomes hard to distinguish between them, especially if they were performed at the same location. Furthermore, it is not possible to determine if a change such as a method addition has been performed at once or in a series of incremental steps in the same session, if only the start and the end point are known. The exact sequence of changes is *lost*.

A further consequence for evolution research is these two effects reinforce each other with the practice of version sampling. Text-files imply a heavy pre-processing: Often, not all versions of a system are taken into account for analysis, but only a fraction. Hence the size of the deltas between the retained version grows much larger. Sampling is a common enough practice that it was included as a requirement of Kenyon[1], a framework for evolution analysis.

3 Change-based Software Repositories

In a change-based repository the history of a software system is not viewed as a sequence of versions, but rather as the sum of *change operations* which were necessary to bring the system to its actual state. These operations can not be deduced with enough precision by differencing two arbitrary program versions [10]. Instead they are recovered by monitoring the IDE usage of programmers while they are building the software. Building a change-based repository involves the following steps:

Program representation. How we represent a software system in a change-based repository to match the problem domain as closely as possible.

Change operations. What constitutes a change operation on a software system, and how to abstract from the lower-level details when they are not needed.

Data retrieval. How to retrieve change operations from an evolving system, using an IDE.

3.1 Program Representation

Our repository stores programs as domain-specific entities rather than text files. Since we focus on object-oriented programs, the entities we store and analyse are object-oriented constructs such as classes and methods, not files and lines. Entities of the problem domain are first-class citizens that can be directly accessed and interacted with, rather than being first parsed, then pre-processed.

Our approach thus represents systems as evolving abstract syntax trees (AST) of their object oriented programs. At various levels of the AST, we find packages, classes, methods, variables and finally statements nodes, as shown in Figure 1. A node *a* is a child of a node *b* if *a* contains *b* (a superclass is not the parent of a subclass, only packages are parents of classes). Nodes have *properties*. These properties vary with the type of node being considered, such as: for classes, name and superclass; for methods, name, return type and access modifier (public, protected or private, if the language supports them); for variables name, type and access modifier.

This abstract syntax tree represents one state of the program, *i.e.,* one state the program went through during its evolution. Each entity also has a *change history* containing all change operations applied to it during the system's evolution.

3.2 Change Operations

Change operations represent the actual evolution of the system under study: They are actions a programmer per-

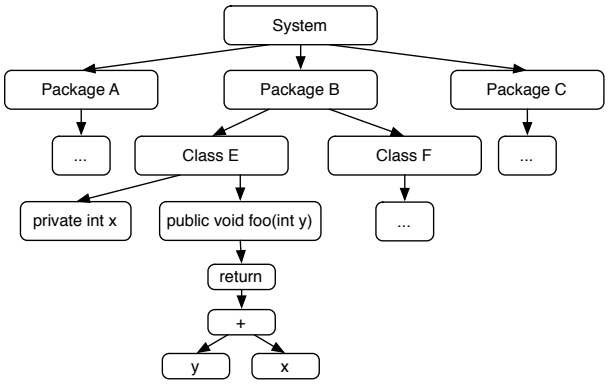

Figure 1. Programs are represented as abstract syntax trees

forms when he changes a program, which are captured, reified and stored in our repository. Change operations enable us to transition from one state of the evolving system to the next. Some examples of change operations are: adding a class to the system, removing a method, moving a class from one package to another, changing the implementation of a method, or performing a refactoring. We support to kinds of operations: atomic and composite change operations.

3.2.1 Atomic Change Operations

Since we represent programs as abstract syntax trees, change operations are, at the finest level, operations on the program's AST. These operations consist of creating, adding, or removing a node in the AST, as well as changing properties of a node:

Creation: creates a node *n* for an entity of a given type *t*.

Addition: adds a node *n* as a child of a given parent *p*.

Removal: removes a node *n* from the children of its parent *p*.

Property change: changes the value *v* of a property *p* of node *n*.

Atomic change operations are executable. By iterating on the list of changes we can generate all the states the program went through during its evolution. In that sense, these operations are sufficient to model the evolution of programs.

3.2.2 Composite Change Operations

Even if atomic change operations are enough to model the evolution of programs, the finest level of granularity is not always the best suited. Representing the entire evolution of a system only by its atomic modifications would lead to an overwhelming mass of information. An abstraction mechanism is necessary. Hence change operations are composable, and can be grouped into higher-level change operations associated with a more abstract meaning. A composite change is simply a sequence of lower-level changes. Executing it amounts to executing the lower-level changes it contains in order. For example, *moving* a class from one package to another consists in first removing it from the old package, and then adding it to the new package. These two change operations can be grouped in a single, higher-level *move class* change operation.

We can see the change-based history of a system as a tree. Its leaves are the fine-grained change operations mentioned above, while the root node represents the entire history of the system, as one change operation. In between are several levels of increasingly coarser-grained change operations:

Developer-level actions: This set of changes constitute a unit from the developer's point of view [1]. For example, changing the definition of a class, adding or changing a method are such changes. Developer-level actions can contain several atomic changes: A method addition contains changes related to the creation and the addition of the program statements within it.

Refactorings: Refactorings [7] are behavior-preserving code transformation aimed at improving the design of source code. The *rename method* refactoring involves changing the method's name, and also all the references from the old method name to the new one. These developer-level actions can be grouped in a higher-level entity representing the refactoring itself. In the same fashion, the *extract method* refactoring replaces a section of code from a method with a call to a newly created method containing this code fragment. These two changes can also be grouped to form a higher-level change.

Bug fixes: A bug fix would consist of all the changes necessary to fix a given bug being grouped and labelled as such.

Development sessions: This change aggregates all the changes done during a single development session by a developer, be they bug fixes, refactorings or developer actions. This is the closest in term of size with a commit extracted from a versioning system.

[1] Note that some of these actions can be performed automatically by IDE tools – such as generating accessor methods – but are still classified at this level.

Features: This higher-level abstraction would be composed of all the lower-level changes which were necessary to build a given feature of the system.

Developer-level actions, refactorings performed by refactoring tools, and development sessions are detected automatically. We have not yet attempted to recover other changes (manual refactorings, bug fixes, features) in an automated way, but plan to. With such a decomposition of the evolutionary data, the user can explore it by "drilling down" to interesting parts, using interactive tools. He or she can start from a high-level view of the entire system's history and arrive to a lower-level view focused on the evolution of a smaller set of closely-related classes, in a given period of time.

In the case study of the article we focus on some of these higher-level changes that have been already implemented in our prototype: developer-level actions, refactorings and development sessions.

3.3 Data Retrieval

Viewed from the mindset of a versioning system user, there is no reason why the approach we propose should get more precise data if the same strategy is used. This is correct. Capturing a more accurate evolution of the system, including refactorings, with a good amount of precision requires a new data source. Hence the data repository we devised does not rely on versioning system data, but on the interaction with an Integrated Development Environment (IDE).

The strategy used by versioning systems, which is to wait for the developer to submit his or her changes to the repository, does not capture the information necessary to build accurate change operations. The core issue is that versioning systems let the developer notify them of changes, instead of taking a more proactive approach. We can adopt such an approach by using an IDE, which can be open to external contributors by means of a plug-in mechanism [16] [18]. Examples of open IDEs includes Squeak[2] and Visualworks[3] for the Smalltalk language, Eclipse[4] and IntelliJ IDEA[5] for the Java language.

IDEs maintain a internal high-level representation of the code they manage. They exploit it to offer several language-dependent tools to increase programmer productivity, such as abstracted source code views – beyond the normal source-level, text based view –, refactoring and autocompletion. They also feature event mechanisms which allow the IDE to react to the programmer's activity.

[2]http://www.squeak.org
[3]http://smalltalk.cincom.com
[4]http://www.eclipse.org
[5]http://www.jetbrains.com/idea

These mechanism can by exploited by tools to closely monitor a system's evolution, with the following advantages:

- User activity can be processed *as it happens*. Large deltas between versions is what makes it hard to distinguish between changes. On the contrary being notified of individual changes means that the deltas are as small as possible, giving much more context for each change and easing the tracking of entities. If an entity is renamed, and is then heavily changed, it is much easier to keep track of it if the changes are processed separately rather than together.
- Time stamps can have an up to the second precision, whereas in a versioning system's repository only the time stamp of the transaction is kept.
- It is possible to be notified of a variety of events to make the analysis more precise. For example refactoring tool usage can be monitored, as well as code navigation or execution errors. If an interface to a versioning system is present in the IDE, it can be monitored too. This allows to easily delimit the stream of change operations in sessions.

3.4 Implementation

We have implemented our approach in an IDE plug-in for the Squeak Smalltalk environment, under the moniker "SpyWare". SpyWare currently monitors the activity of the IDE user and store code modifications in a change-based repository, alongside other useful information, such as refactorings or navigation paths through the code. SpyWare also allows to exploit this information through interactive data reports and visualizations. It also can regenerate the code of the project at any given point in time, and offers dedicated code browsers for this task. More information, screen captures and a web-based demo can be found at `http://romain.robb.es/spyware.html`.

4 Case Study: Applications of Change-Based Mining

We have already performed some preliminary case studies on several student projects we monitored with an earlier version of our prototype [14] [13], but without considering refactorings, as students did not use them. We focus on the relation between development sessions and refactorings in the following case study.

4.1 Refactoring Detection

Locating refactorings is important since it permits to characterize phases of development of a system, a useful

tool to assess a system's evolution. Weissgerber *et al.* list other uses in [17], including detecting errors and replaying changes.

Several works perform refactoring detection based on conventional software repositories [17] [3] [4]. They analyse successive versions of a software system in order to detect which refactorings have been employed between versions. The most recent approaches, Weissgerber *et al.* and Dig *et al.* still feature several restrictions:

- The focus is on refactorings which change the interface of classes, such as *rename method*, *push up method* or *add parameter*. Other refactorings operating at the method body level are not detected (*extract method*, *rename temporary* , *etc.*).
- These approaches work best when the entities are not modified by other changes in the same session. Weissgerber and Diehl use clone detection to match method bodies in case of renames, but a method which is both renamed and has a portion of its code removed using *extract method* would be harder to detect.
- In the same way, if a method body is modified by non-refactoring operations before or after a refactoring, the detection of the refactoring is more difficult.
- Weissgerber looks for comments in the CVS log file to find areas where refactorings were performed and empirically assess its precision, while Dig bases himself on release notes. It is not possible to know which refactorings happened and accurately measure the precision of the approaches: The lookup is manual.

4.2 Case Study Questions

The main problem of refactoring detection approaches is the noise induced by other changes performed in the same locations as the refactorings. Our approach is not affected by this since it leverages the IDE to record refactorings performed via tools rather than attempting to detect them.

This refactoring log combined with our general change-based model allow us to empirically study when and how refactorings are actually applied. We test if situations when refactorings are performed on heavily changing code – the hardest one to detect – happen frequently in practice.

Specifically, our case study answers the following questions:

- What kind of refactorings were used? Are they interface-changing refactorings detected by other approaches?
- Were several refactorings applied to the same entities in the same session, making them harder to detect?
- Were refactorings performed alongside other changes, complicating their detection in the process?

We study two different systems. The first is our prototype itself. It has been monitoring itself since its inception, and can track refactorings applied to it since a few months. The other case study comes from an independent developer who allowed us to monitor him on one of his projects for several months. It is a web application which we will refer to as "Project X" for privacy reasons.

4.3 Types of Refactorings Performed

Table 1 lists the different types of refactorings performed on the two systems. The first part of the table lists refactorings which have a large impact on the interface of the class, such as changing a method signature or renaming a method. The second part lists refactorings which have a little impact on class interfaces (adding or removing a single method to a class), while the third part contains refactorings which do not change the interface of a class, only its implementation.

Type of Refactorings	Spyware	Project X
Add Parameter	6	0
Push Up Method	5	45
Rename Class	3	0
Rename Method	10	0
Abstract Instance Variable	0	1
Push Up Instance Variable	0	5
Total	24 (28%)	51 (54%)
Extract Method	37	0
Extract Method to Component	2	0
Inline Method	1	0
Total	40 (47%)	0 (0%)
Extract Expression to Variable	6	0
Inline Temporary	3	0
Rename Instance Variable	4	1
Rename Temporary Variable	5	42
Temporary to Instance Variable	3	0
Total	21 (25%)	43 (46%)
Total Number of Refactorings	85	94

Table 1. Types of refactorings performed

The refactorings detected by Weissgerber *et al.*, or Dig *et al.*, are all in the first category. We see that the first category constitutes 30% of the refactorings in SpyWare, whereas method-level refactorings refactorings (second or third category) constitute 70% of the refactorings. Project X is different: no methods were renamed or extracted. A majority of refactorings were either *push-up method* or *rename temporary*. A large half of the refactorings were interface-level refactorings, the being at the method-body level. It is interesting to notice that a few marginally popular refactorings in study 1 are overwhelmingly used in study 2: Refactorings habits seem to vary wildly among developers.

4.4 Multiple Refactorings During Sessions

Refactorings applied in large numbers over a single development session are interesting since they carry the risk that multiple refactorings are applied to the same entity.

Table 2 shows result in agreement between the two studies: a majority of sessions do not feature refactorings at all. In the rest, twice as many sessions feature a low number of refactorings, but the ones having a large number of refactorings have the majority of refactorings performed, between 59 and 80 percent of them.

Number of refactorings per session	Spyware	Project X
0	63 (72%)	111 (87%)
1	7	5
2-4	10	5
Total	17 (20%)	10 (8%)
Total Refactorings	35 (41%)	19 (20%)
5-9	5	3
10 - 19	2	1
20 - 29	0	2
Total	7 (8%)	6 (5%)
Total Refactorings	50 (59%)	75 (80%)
Total number of sessions	87	127
Total number of refactorings	85	94

Table 2. number of refactorings per session

4.5 Refactorings and Method Modifications

We computed how often a method was involved in several refactorings and found differing results for each case study. For Project X, no method was involved in several refactorings. In our prototype however, we found 28 occurrences of these: Most were affected only two or three times, but a few methods were affected by four or five refactorings. Sessions tended to have several methods associated with several refactorings. Refactorings such as *extract method* and *extract expression to temporary* often were applied several times to one method, as well as inlining refactorings: 15 times in total.

We also computed how often a method was refactored and also changed by other means in our two studies. The results are summed up in Table 3. A method was considered heavily modified if it was modified (*i.e.,* recompiled) more than 3 times during a session outside of refactorings.

Table 3 tells us that: (1) a sizeable amount of methods were involved in refactorings but were modified, rendering the detection of this fact more difficult. Less than half of the methods were only refactored in SpyWare, and nearly 30% changed heavily. Project X is more positive: three-quarters of methods were refactored without other modifications. This is due to the high number of *push-up method* refactorings, whereas our project contains much more *extract method* than *push up method* refactorings. (2) two-third to 80% of methods involved in refactorings were created in the same session: Without preliminary data on the method, it is much more difficult to know if it was involved in a candidate refactoring.

Method Evolution Characteristic	Spyware	Project X
Changed Overall	590	1587
Involved in Refactorings	150	62
Involved and Not Changed	72 (48%)	47 (76%)
Involved and Changed	35 (23%)	7 (11%)
Involved and Heavily Changed	43 (29%)	8 (13%)
Involved and Created	101 (67%)	50 (80%)

Table 3. Number of refactorings per method per session

4.6 Conclusions

In a project where refactorings are at the interface-level – such as Project X –, it seems much easier to detect refactorings, corroborating the results found in practice by other approaches. However, when refactorings at the method body level occur – such as in SpyWare –, they tend to come in groups and cause more body-level modifications, rendering them much harder to detect. Incidentally, they are not the focus of most refactoring-detection approaches. As for the usability of our repository, its accuracy and change-based model allowed us to query the data in a straightforward way, as shown in the high-level data we extracted. We think a change-based representation supporting composition is well suited for this type of analysis.

5 Discussion

This section discusses our contribution according to: (1) the advantages and drawbacks of our approach with respect to classical and popular versioning systems, namely CVS and SubVersion; (2) its relationships with the general field of software configuration management research, and particularly change-based versioning systems; and (3) a discussion of the findings of our case study.

5.1 Change-based versus snapshot-based repositories

Our approach has the following characteristics:

It is Domain-specific: We treat object oriented concepts such as classes and methods as first-class citizens. No pre-processing is necessary to recover these entities. It is hence easier to reflect at the level of classes and methods. On the other hand, our approach is harder to adapt to other programming languages and IDEs since a significant porting effort has to be undertaken.

Less information loss: Each change has a precise time stamp information up to the second, whereas change information in a versioning system only keeps the time stamp of the actual commit to the repository. This allows us to recover the exact sequence of changes employed during each development sessions. In the same fashion, monitoring an IDE enables us to record events such as refactorings rather than detecting them.

Changes are first-class citizens: Changes are explicitly modelled and easily accessible. An abstraction mechanism allows changes to be considered at several levels. In the case studies of this paper we primarily chose the abstractions of development sessions and refactorings as basis for our work.

Rare case studies: Since the information stored by our approach was previously discarded, we can not use the same case studies as other approaches. This is a major drawback which limits us to a few case studies and future project. So far we have monitored 9 student projects, our prototype and Project X. We hope to expand our pool of possible projects soon. This requires a port of our tools on more popular platforms, such as the Eclipse/Java platform.

Memory and speed requirements: Our approach maintains a fine-grained model of a system's evolution. Its requirements in terms of processing power are thus higher than other, lightweight approaches. Performance has not been a concern so far, so it is not optimized. Still, we need to test our approach with larger systems to know how it scales.

Non-code repositories: Bug databases and e-mail archives are also important sources of information we have not discussed. We have not yet researched how to link then with the information found in our repository.

5.2 Relations to Software Configuration Management

One could think we are trying to implement a new kind of versioning system to support our task. Although we share several concepts with the SCM community, we are not building a SCM *per se*, for the following reasons:

- SCM systems do much more than just storing data about a software system. They must detect conflicts between versions, provide configuration selection, handle derived versions and much more. We are only interested in modelling the evolution of a system at a fine-grained level.

- The main drawback of our approach is the lack of case studies. Providing an SCM replacement would add a barrier to entry and discourage potential users: Developer tend to keep their SCM choices for a long time. CVS has been in use for 15 years now, and starts only now to be replaced by SubVersion, mainly because SubVersion is an improvement over CVS but is still largely compatible. We see ourselves as complementing rather than replacing current versioning systems.

[2] details the sub-field of change-based versioning systems. These range from conditional compilation, to full-featured systems such as EPOS[8]. The idea is to build a system from change sets rather than versions of objects. Considering change sets as first class citizens permits much richer configuration possibilities. However the change granularity they consider is the feature level, which is much coarser than ours. These systems also adopt the checkin/checkout philosophy of state-based versioning systems and so have the same information loss problem. An exception is operation-based merging [11], which explicitly stores change operations, but is very theoretical and does not tell where or how to get these operations, or what they consist of. Moreover its focus is explicitly software merging: It lacks features to support program comprehension through evolution.

5.3 Refactoring Case Study

Henkel *et al.* [9], Ekman *et al.* [5] also record refactorings. In addition, the idea of refactoring logs is being adopted by mainstream IDEs: Frameworks to support these activities are developed for Eclipse. We are however the only one to our knowledge to include refactorings in a more general framework aimed at modelling in detail the evolution of software systems.

Our limited amount of case studies means that we cannot reproduce the conditions of the experiment of Weissgerber and Diehl. A direct comparison is not possible since the precise information our approach requires was not recorded. We can only perform an approximate comparison by using alternative case studies, and comparing the usage patterns of these studies. This raises the questions of the relevance of our comparison.

However, we believe the figures we have stress the limits of refactoring detection in software archives. In the case studies we took, several refactorings were undertaken in

parts of the system which were heavily modified in the same session, rendering further detection hard. As the habits of using refactoring tools becomes commonplace, such behavior might increase, making the task difficult to accomplish without IDE help. Even if people do not use our full-fledged approach, we believe the use of refactoring logs can be a valuable insight to refactoring detection tools and provide an objective benchmark to measure tool accuracy.

Refactoring detection approaches have a sizeable advantage over ours: they can detect refactorings performed manually, something we cannot do at the moment. A possible extension of our work is to detect manual occurrences of refactorings as well. We believe that analysing sequences of changes can yield some interesting results in this respect, but have yet to validate this theory.

6 Conclusion

In this paper we presented the principles behind a change-based repository and how they can address some of the limitations of more classic software repositories to perform software evolution. By monitoring IDE usage instead of relying on the checkin/checkout model of versioning systems, a change-based repository can capture more accurate information about an evolving system, and represent its evolution as a sequence of changes rather than several successive versions. We implemented our ideas and used one of these repositories to assess how often situations in which refactoring detection tools – based on versioning system repositories – have problems really occur.

Our work can be extended in the following ways: (1) detecting manual occurrences of refactorings alongside those performed with automatic tools; (2) we want to study in more details what happens during a development session in order to characterize them; (3) we want to widen the usage of our tool and the case studies available by promoting it and porting it to more popular developer platforms.

References

[1] J. Bevan, J. E. James Whitehead, S. Kim, and M. Godfrey. Facilitating software evolution research with kenyon. In *Proceedings of the 10th European software engineering conference*, pages 177–186, 2005.

[2] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, June 1998.

[3] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, 2000. Also appeared in ACM SIGPLAN Notices 35 (10).

[4] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

[5] T. Ekman and U. Asklund. Refactoring-aware versioning in eclipse. *Electr. Notes Theor. Comput. Sci.*, 107:57–69, 2004.

[6] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, Oct. 2005.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[8] B. Gulla, E.-A. Karlsson, and D. Yeh. Change-oriented version descriptions in epos. *Softw. Eng. J.*, 6(6):378–386, 1991.

[9] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *Proceedings International Conference on Software Engineering (ICSE 2005)*, pages 274–283, 2005.

[10] M. Kim and D. Notkin. Program element matching for multi-version program analyses. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 58–64, 2006.

[11] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.

[12] R. Robbes and M. Lanza. Versioning systems for evolution research. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 155–164. IEEE Computer Society, 2005.

[13] R. Robbes and M. Lanza. An approach to software evolution based on semantic change. In *Proceeding of FASE 2007*, page to appear, 2007.

[14] R. Robbes and M. Lanza. A change-based approach to software evolution. In *ENTCS volume 166, issue 1*, pages 93–109, 2007.

[15] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 106–110, Los Alamitos CA, 2004. IEEE Computer Society Press.

[16] D. Čubranić and G. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.

[17] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 231–240, 2006.

[18] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.

0-7695-2950-X/07 $20.00 © 2007 IEEE