Change-based Software Evolution

Romain Robbes and Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Abstract

Software evolution research is limited by the amount of information available to researchers: Current version control tools do not store all the information generated by developers. They do not record every intermediate version of the system issued, but only snapshots taken when a developer commits source code into the repository. Additionally, most software evolution analysis tools are not a part of the day-to-day programming activities, because analysis tools are resource intensive and not integrated in development environments. We propose to model development information as change operations that we retrieve directly from the programming environment the developer is using, while he is effecting changes to the system. This accurate and incremental information opens new ways for both developers and researchers to explore and evolve complex systems.

1. Introduction

The goal of software evolution research is to use the history of a software system to analyse its present state and to predict its future development [11, 5]. Such an analysis needs a lot of information about a system to give accurate insights about its history. Traditionally researchers extract their data from versioning systems, as their repositories contain the artifacts the developer produce and modify.

We argue that the information stored in versioning systems is not complete enough to perform higher quality evolution research. Since the past evolution of a software system is not a primary concern for most developers, it is not an important requirement when designing versioning systems. They favor features such as language independence, distribution and advanced merging capacities.

We need to prove to developers that results in software evolution research are immediately useful to them by improving the integration of our tools in their day-to-day processes. Most tools are tailored for an use "after the fact", once the main development is over and before a new feature is added. A common approach is to download several versions from a repository and to process them all at once. This shows that incremental processing is limited, and computations are long and resource-intensive. We need to provide more incremental, lightweight approaches that developers can use in their work.

This paper presents our approach to tackle both problems of accurate information retrieval and developer use of evolution tools. We believe the most accurate source of information is the Integrated Development Environment (IDE) the developers are using. By hooking our tools into an IDE, we can capture evolution information as it happens, treat it in an incremental manner, and interact with the environment to improve the usability of our tools. Our approach is based on a model of the changes developers are applying to the system and hence treats changes as first-class entities. In that sense, we do not make a distinction between the system and the changes that are performed on it, *i.e.,* software engineering *is part of* software evolution.

Structure of the paper: Section 2 expands on the nature and consequences of the problems we presented. Section 3 introduces our alternative approach. Section 4 describes the implementation status of SpyWare, our prototype. Section 5 describes how such a model can be used in practice and how problems are stated and solved differently in an incremental, change-based world. Section 6 concludes the paper.

2. Current Approaches to Software Evolution

To perform efficient evolution research, accurate data about the system under study is required. Despite this need, the tools the community uses to gather the data do not provide such accurate information. At the core of most data recovery strategies is the versioning system used by the developers of the system.

The main criteria in choosing a versioning system to extract data from is how many systems it versions, especially open-source ones: developers allow free access to their repositories. The largest open-source software systems (Mozilla, Apache, KDE, etc) use either CVS or Subversion, researchers therefore write their tools to gather data from these repositories.

2.1. Limitations in Information Gathering

In a previous study [12] we showed that most versioning systems in use today (including CVS and Subversion) are indeed losing a lot of information about the system they version. We identified two main, orthogonal, reasons: most versioning systems are (1) *file-based*, and (2) *snapshot-based*.

File-based systems. Most of these systems still function at the file level, as this guarantees language independence. On the other hand, it involves extra work to raise the level of abstraction to the programming language used by the system [14, 9], because the collected information is obfuscated:

- The semantic information about a system is scattered in a large amount of text files: there is no built-in central repository of the program structure, it has to be created manually.
- Keeping track of a program-level (not text-level) entity among several versions of the system is hard since it involves parsing several versions of the entire system while taking into account events such as renames of files and entities due to refactorings. Hence some analyses are performed on data which has been sampled [7, 8]: only a subset of the versions are selected because of time and space constraints. This increases the changes between each versions, and makes it harder to link entities across versions since the probability they have changed is higher. Other analyses do without the parsing of the files altogether, basing themselves on coarser-grained information such as number of lines or size of directories [4, 6].

Snapshot-based systems. Changes between successive versions of the software are stored on explicit requests (called *commits*) by the developer. The time between two developer commits varies widely, but is often on the order of several hours or days. What happens between two commits is never stored in the versioning system, and we have to deal with degraded information:

- Since commits are done at the developer's will, several independent fixes or feature additions can be introduced in one single commit, making it hard to differenciate them.
- The time information of each change is reduced to the time when a commit has been performed: beyond the task of extracting the differences between two versions of the system, all information about the *exact sequence of changes* which led to this differences is lost.

2.2. Practical Impacts of Information Loss

Example. The example in Figure 1 shows how this loss of information can significantly degrade the knowledge we

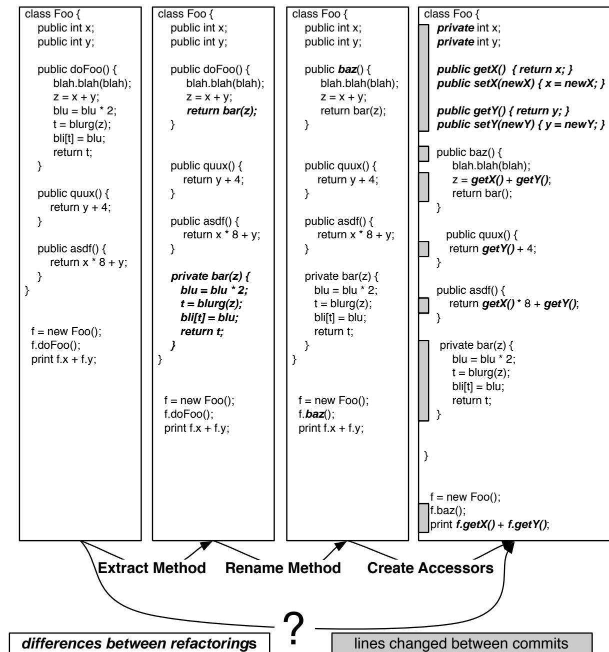

Figure 1. Simple refactoring scenario leading to evolution information loss.

get of a system. In this simple scenario a developer starts a short refactoring session, in which he refactors the method *doFoo*. He (1) extracts a block of statements in a new method *bar*, (2) replaces direct accesses to instance variables *x* and *y* with accessors throughout the entire system, and (3) renames *doFoo* to *baz*, replacing all references to *doFoo* in the code base.

He then commits these changes. This is a very small commit, less than a minute of work, since in current IDEs all these refactoring operations can be semi-automated. Commits usually imply larger change sets than this simple example. According to the information gathered from the versioning system, the following physical changes happened:

- The method *doFoo* changed name and is now significantly shorter. This makes it hard to detect if the new method *baz* is really the same entity that *doFoo* was. A simple analysis would conclude that method *doFoo* disappeared.
- There are several new methods: *bar*, *baz*, and accessor methods *getX*, *getY*, *setX*, *setY*.
- Several methods had their implementation modified because of the rename of *doFoo* and the introduction of accessors, possibly scattered among several files of the entire codebase.

In this example, only refactorings – by definition behavior-preserving[3] – have been performed. The logical changes to the system are trivial, yet this commit caused many physical changes: Its importance measured in lines of codes is exaggerated. Without a sophisticated, time-consuming analysis [14], some entities such as *doFoo* are lost, even if they still exist in the code base. On the other hand, using such a time-consuming analysis makes it harder to integrate our tools in day-to-day activies.

Moreover, the simple scenario depicted above assumes that a developer commits after every couple of minutes of work. In reality, it is more on the order of hours. The change amount would be greater, and changes would be even more diluted and less recoverable.

2.3. The Lack of Integration

The way we collect our information has shaped our tools to function likewise. The typical procedure to fetch information out of a version repository is to (1) download a set of versions from the repository, (2) build a program representation for each of the versions, and (3) attempt to link successive versions of entities.

This approach is clearly only suited for an off-line activity, because even if sampling is used it is time-consuming (hours or days to complete on a large-scale system). Currently, forward and reverse engineering are two very distinct, separate activities. When applied in practice, reverse engineering is performed by specialized consultants acting on unknown systems under time constraints.

To better accomodate developers, software evolution tools need to be incremental in nature and easily accessible from IDEs. Tools need to focus on smaller-scale changes, when developers are working on smaller parts of the system, as well as providing a "big picture" view of the system to external people such as project managers.

All these necessities become even more important with the advent of agile methodologies such as extreme programming (whose motto is "embrace change" [1]), which advocate continuous refactorings and changes in the code base.

2.4. Ideas Behind our Approach

Our approach, presented in the next sections, stems from the following observations:

- Versioning systems are not a good source to retrieve information, as they store changes at the file level. They also store changes at commit time, yelding too coarse-grained changes.
- More and more developers are nowadays using IDEs, featuring a wealth of information and tools, making development more effective and increasing the change rates of systems.
- For evolution tools to gain acceptance, they must (1) adapt to this increase of the rate of change, (2) be used by the developers themselves as part of their day-to-day activities, (3) be able to focus on small-scale as well as large-scale entities, and (4) support incremental updates of information, as day-long information retrieval phases are a serious flaw for daily usage.

3. An Alternative Approach to Evolution

Our approach is based on two concepts: (1) an IDE integration to record as much information as possible and to allow easy access to our tools, and (2) a model based on first-class change operations to better match the incremental process of developing software.

3.1. Using the IDE as Information Source

Most programmers use IDEs for their day-to-day tasks, because they are powerful tools featuring support for semi-automatic refactoring, incremental compilation, unit testing, advanced debugging, source control integration, quick browsing of the system, etc. Most of them are extensible by plug-in systems.

IDEs are able to do so much because they have an enormous amount of information about the developer and his system. Being able to browse or refactor the system already implies having a reified program model. Thus we advocate integrating our tools in an IDE, and using the IDE itself as the source of evolution information instead of the versioning system. Tool integration increases tool visibility and is a first step to feature them in the developer's workflow. To use the IDE as the source of information is the closest we can get to understand the developer's intentions.

Most IDEs feature an event notification system, so tools can react to what the developer is doing. Hooks monitor when a class is compiled, or when a method is recompiled. The approach we propose use these IDE hooks to react when a developer modifies the system by creating data defined as *first-class change entities*.

3.2. First-class Change Entities

First-class change entities are objects modeling the history of a system following the incremental way it was built. They contain information to reproduce the program of which they represent the history. When executed, they yield an abstract representation of the program they represent at a certain point in time. They also contain additional information interesting for evolution researchers, such as when and who performed which change operations.

Traditional approaches model the history of a program as a sequence of versions. This is memory-consuming, since most parts of the system do not change and are simply duplicated among versions. This is why most approaches include a sampling step, aimed at reducing the number of versions by selecting a fraction of them. This sampling step hence increases the changes between successive versions, rendering fine-grained analysis even harder. In constrast, our approach only stores the program-level differences between versions, and is able to reproduce the program at any point in time.

Change operations also model with greater accuracy the way the developer thinks about the system. If a developer wants to rename a variable, he does not think about replacing all methods referencing it with new methods, even if that is what the IDE ends up doing: Modeling incremental modifications to the system eases its understanding.

Although we model program evolution with first-class change operations to ease reverse engineering, we believe it is useful for forward engineering as well. Most end-user applications feature an undo mechanism, but most program editors do not provide a sensible one at the semantic level. First-class change operations could enable this, hence facilitating exploratory programming by trial and error. First-class change entities can also ease arbitrary program transformation to facilitate program evolution, following the same scheme as semi-automated refactorings[13].

To sum up, our approach consists of the following steps:

1. Use the hooks of the IDE to be notified of developer activity.
2. React to this activity by creating first-class change objects representing the semantic actions the developer is performing.
3. Execute these change objects to move the program representation at any point in time.

Advantages. The advantages of this alternative approach over gathering data from a repository and performing offline analysis are the following:

- *Accuracy*. Reacting to events as they happen gives us more accurate information than the one stored in the versioning system. Timestamps are more precise, not reduced to commit times. Events happen one by one, giving us more context to process them than if we had to process a batch of them, originated from an entire day's work.
- *Incrementality*. It is significantly easier to maintain a semantic representation of the model. Events originating in the IDE are high level. Their granularity is the one of classes and methods, not files and lines. Code parsing is required only at the method level.
- *Fine-grained*. Every program entity can be modelled and tracked along its versions, down to the statement level if necessary. There is no state duplication, leading to space economies when an entity does not change during several versions.
- *Flexibility*. Going back and forward in time using change objects is easy. It leads to more experiments with the code base, easing "trial and error" in development.

Drawbacks. We have identified possible issues and implications with our approach:

- *Acceptance.* Evolution researchers use CVS despite its flaws, because it is the versioning systems most developer use. Subversion is a newer versioning system gathering momentum because it is close enough to CVS. Hence to be succesful we need to depart from people habits as less as possible.
- *Validation.* Our approach needs to be evaluated with case studies. We are monitoring our prototype itself, but without a "real-world" case study we are unsure about performance constraints. Our approach works best for new projects. This limits possible case studies.
- *Paradigm shift.* Such an incremental approach to various problems needs new tools and new practices to be defined.
- *Applicability.* Our approach is language-specific, which involves more effort to adapt it to a new language than conventional file-based approach. However, our current prototype implementation is split into a language-independent part and a language-dependent one. Only the latter one must be adapted to other languages/IDEs.

To address acceptance issues, we can integrate our tools in mainstream IDEs, such as Eclipse, which features a plugin mechanism. The monitoring part of the system is not intrusive and is not visible to users. Keeping track of the data across sessions or programmer locations can be done by creating a "sync" file which would be part of the current project. The versioning system itself would be used to broadcast and synchronize information.

4. Our Prototype: SpyWare

Our ideas are implemented in a prototype named SpyWare (see Figure 2), written in Squeak [10]. It monitors developer activity by using event handlers located at IDE hooks, and generates change operations from events happening in the Squeak IDE. Changes supported so far are shown in Table 1.

Change Type	Package	Class	Method	Variable	Statement
Creation	X	X	X	X	X
Addition	X	X	X	X	X
Removal	X	X	X	X	X
Rename	-	X	-	-	-
Superclass Change	no	X	no	no	no
Property Change	X	X	X	X	X
Refactoring	-	-	-	-	-

Table 1. Changes supported by SpyWare.

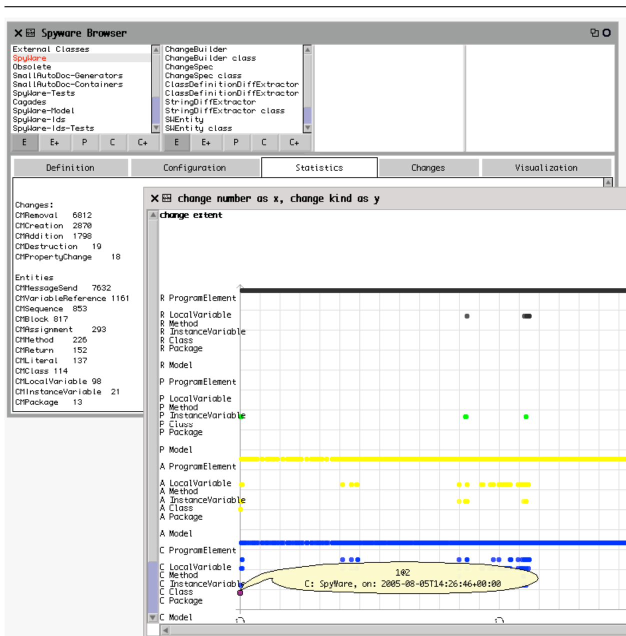

Figure 2. SpyWare's UI features browsing, statistics and interactive visualizations.

SpyWare associates these change operations to program entities up to the statement level. It is possible to track changes to a single statement. Entities are uniquely identified independently from their name: A rename is a trivial operation. SpyWare can also generate the source code of the program it is monitoring at any point in time, by applying or reverting change operations. It also features basic support for interactive visualizations of the history.

Our future work includes the definition and detection of higher-order changes such as refactorings, or distinct features of the monitored program, out of the basic building blocks we already defined. SpyWare is currently single-user: we plan to make it multi-user soon.

5. Change-Based Software Evolution

We believe our approach has the potential to address several problems in both reverse and forward engineering, as an IDE integration makes the dialogue between the two activities more natural.

Facilitating program comprehension. Processing finer-grained changes will allow us to detect and characterize changes with greater accuracy. Detecting and keeping track of all the refactorings performed on the code will allow us to track specific entities with more accuracy. We also believe that it is possible to characterize changes as either bug fixes, refactorings or feature additions and that this information will allow to focus analysis on specific changes by contextualizing them.

Our model allows us to characterize or classify changes and entities in arbitrary ways (using properties or annotations). This facility can be used to ease understanding of the code as well. Contrary to classical versioning systems where branches are fixed and are set up before modification, our model permits the modification of properties of changes while reviewing them. Changes that need to be grouped can be tagged for an easier handling.

Recording the complete history of a system allows for fine-grained understanding of a dedicated piece of code by reviewing its introduction and modifications in context of surrounding modifications, *e.g.,* it is useful to know whether a line is present from the beginning of a method or much later because of a bug fix.

Facilitating program evolution. First-class change objects can be broadcasted through a network to increase awareness and responsiveness to changes, by providing developers insights of what other developers are doing. Such a system would tell them if their changes are conflicting with other people's changes interactively. This will help avoiding long and painful merge phases.

Change-based operation coupled with entity-level tracking will ease refactoring, *e.g.,* in our current model, the name of an entity is just a property: A rename does not affect identity.

Merging reverse and forward engineering. Higher-level languages and tools promote a faster and easier implementation of functionality, which translates into a higher

change rate of the system. Hence some reverse engineering activities need to be done on a smaller scale, but with a higher frequency and accuracy, to keep track of what has been done in the system before resuming work on a part of the system.

Change operations between two versions of the system can be used to generate an automatic and interactive change log to bring other developers up to speed on the changes a developer made.

6. Conclusion

Software evolution research is restrained by the loss of information which are not captured by most versioning systems. Evolution analysis tools are not used by developers because they are not integrated in an IDE and require time-consuming data retrieval and processing phases. They are not suited for smaller-scale, day-to-day tasks [2].

We presented an alternative approach to gather and process information for software evolution. We gather data from the IDE the developer is using rather than the versioning system. We model program change as first-class entities to be closer to the developer's thought process. Changes can manipulate the model to bring it at any point in time in a very fine-grained way.

Our approach being incremental, fine-grained and integrated in an IDE, we consider it is suited for a daily use by developers. To validate our hypothesis, we are currently implementing a prototype named SpyWare.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science foundation for the projects "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997), and the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976). We thank Marco D'Ambros and Mircea Lungu for giving valuable feedback on drafts of this paper.

References

[1] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley, 2000.

[2] S. Demeyer, F. Van Rysselberghe, T. Gîrba, J. Ratzinger, , R. Marinescu, T. Mens, B. Du Bois, D. Janssens, S. Ducasse, M. Lanza, M. Rieger, H. Gall, M. Wermelinger, and M. El-Ramly. The Lan-simulation: A Research and Teaching Example for Refactoring. In *Proceedings of IWPSE 2005 (8th International Workshop on Principles of Software Evolution)*, pages 123–131, Los Alamitos CA, 2005. IEEE Computer Society Press.

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

[4] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[5] H. Gall, M. Jazayeri, R. Klösch, and G. Trausmuth. Software evolution observations based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM'97)*, pages 160–166, Los Alamitos CA, 1997. IEEE Computer Society Press.

[6] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

[7] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, 2004. IEEE Computer Society Press.

[8] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[9] C. Görg and P. Weissgerber. Detecting and visualizing refactorings from software archives. In *Proceedings of IWPC (13th International Workshop on Program Comprehension*, pages 205–214. IEEE CS Press, 2005.

[10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*, pages 318–326. ACM Press, Nov. 1997.

[11] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[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] D. Roberts, J. Brant, R. E. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, Apr. 1996.

[14] Q. Tu and M. W. Godfrey. An integrated approach for studying architectural evolution. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 127–136. IEEE Computer Society Press, June 2002.