Of Change and Software

Doctoral Dissertation submitted to the
Faculty of Informatics of the University of Lugano
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Romain Robbes

under the supervision of
Michele Lanza

December 2008

Acknowledgments

As I type this, I have spent four years working in Lugano. Four years is a long time. I was fortunate to share that time with a number of people who made this part of my life exciting.

First of all, many thanks to Michele Lanza, my advisor, for being there all along, starting at Lugano's train station back in October 2004. Waiting for me as I arrived after a long night in the train showed me how much you cared about your students. During these four years, you showed it over and over. I wish all the best to you, Marisa and Alessandro.

Many thanks to Mehdi Jazayeri for starting the Faculty of Informatics in Lugano and for giving me the opportunity to work there. Without you, my life would have been radically different. Special thanks to Mehdi and the other members of my dissertation committee, Stéphane Ducasse, Jacky Estublier, Jean-Marc Jézéquel, Mauro Pezzè and Andreas Zeller, for showing interest and investing a part of your precious time to evaluate my work.

Special thanks to Doru Gîrba for the early discussions that led me toward this subject. Thanks also to Oscar Nierstrasz, and the rest of the Software Composition Group in Bern, for hosting me during the summer of 2004. My stay at SCG certainly influenced me to continue my studies in Switzerland.

Thanks to Damien Pollet, Yuval Sharon and Alejandro Garcia for your collaborations with me. Thanks Damien for the cool research ideas, Yuval for spending much time in the internals of Eclipse when building EclipseEye, and Alejandro for making movies of program histories –I'll watch one over "The Dark Knight" any day. Thanks to Alejandro, Philippe Marschall, and the second promotion of USI Informatics students for allowing me to collect the data I much needed.

Special thanks to the members of REVEAL for being the coolest research group around! Thanks to Marco D'Ambros, Lile Hattori, Mircea Lungu and Richard Wettel for being both extremely competent coworkers, and such a great deal of fun to have around. We worked hard, played hard and had some amazing trips together. Your presence was –as a matter of fact– motivational.

Many thanks to my former flatmates, Cyrus Hall, Cédric Mesnage, Jeff Rose and Élodie Salatko. We shared great times in a small office and a gigantic living room. And the french-american struggle was always entertaining.

Thanks also to the rest of the USI Informatics staff, former and current. Thanks to Laura Harbaugh, Marisa Clemenz, Cristina Spinedi and Elisa Larghi for making my life much easier on the administrative side. Thanks to all the professors, and especially Amy Murphy and Cesare Pautasso for the teaching opportunities. Thanks to Jochen Wuttke and Domenico Bianculli for investing time in the thesis template I am presently using.

Thanks to all the coworkers I generally had fun with, in no particular order (There were too many fun moments to keep count!): Giovanni, Giovanni, Giovanni, Nicolas, Alessandra, Alex, Milan, Morgan, Amir, Mostafa, Anna, Paolo, Vaide, Adina, Monica, Marcin, Dmitrijs, Philippe, Edgar, Navid, Francesco, Francesco, Mark, Avi, Onur, Sasa, Ruben, Lasaro, Alessio, Aliaksei, Daan, Shane, Matteo, Fred, Tom, Julian. Thanks to Thanassis for the fun in Athens, even if the timing was not best for you.

Thanks to my friends back in France. You are too numerous to mention, but you know who you are. Visiting you once every few months was always a refreshment. A special thanks to the ones who had the courage to visit the dangerous city of Lugano: Hurain Chevallier, Denis Meron, Cyril Bazin, Céline Bernery and Étienne Ailloud.

Thanks to the other category of people which are too numerous to mention, the ones I met, had scientific discussions, and had fun with at various conferences. I'm looking forward to meet you again at the next edition of . . . well, you know.

Thanks to my extended family, which also happens to be too numerous to mention. It is always a pleasure to see you, even if it is not often.

And most of all, many many thanks to my family: Isabelle, Didier, Mariette and Benjamin Robbes. Without your unconditional love and support, I could not have concluded this work.

Romain Robbes
October 2008

Abstract

Software changes. Any long-lived software system has maintenance costs dominating its initial development costs as it is adapted to new or changing requirements. Systems on which such continuous changes are performed inevitably decay, making maintenance harder. This problem is not new: The software evolution research community has been tackling it for more than two decades. However, most approaches have been targeting specific maintenance activities using an ad-hoc model of software evolution.

Instead of only addressing individual maintenance activities, we propose to take a step back and address the software evolution problem at its root by treating change as a first-class entity. We apply the strategy of reification, used with success in other branches of software engineering, to the changes software systems experience. Our thesis is that a reified change-based representation of software enables better evolution support for both reverse and forward engineering activities. To this aim, we present our approach, Change-based Software Evolution, in which first-class changes to programs are recorded as they happen.

We implemented our approach and recorded the evolution of several systems. We validated our thesis by providing support for several maintenance activities. We found that:

- *Change-based Software Evolution eases the reverse engineering and program comprehension of systems by providing access to historical information that is lost by other approaches. The fine-grained change information we record, when summarized in evolutionary measurements, also gives more accurate insights about a system's evolution.*
- *Change-based Software Evolution facilitates the evolution of systems by integrating program transformations, their definition, comprehension and possible evolution in the overall evolution of the system. Further, our approach is a source of fine-grained data useful to both evaluate and improve the performance of recommender systems that guide developers as they change a software system.*

These results support our view that software evolution is a continuous process, alternating forward and reverse engineering activities that requires the support of a model of software evolution integrating these activities in a harmonious whole.

Contents

Figures

Tables

Chapter 1

Introduction

Software evolution consists in adapting software to new or updated requirements, and prevent or fix defects. Software evolution causes problems which have no satisfying solution yet –and perhaps never will. We argue that reifying change itself, that is, representing changes as explicit, manipulable entities, gives us more leverage to deal with the problems.

We first describe the problems associated with software evolution. We then motivate why a change-based model of software evolution would be helpful to support software evolution. The intuition behind our thesis is that the process of reification has always been a powerful tool to address software problems, but has not been fully applied to the change process. We present our thesis and the research questions we use to validate it, before giving a roadmap to the remainder of this work.

1.1 The Challenges of Software Evolution

Lehman's laws of software evolution state that as software systems grow and change over time, each further modification is more difficult [LB85]. In particular, a system must continuously change to remain useful in a changing environment (law 1). If nothing is done to prevent it, the system decays: Its complexity increases (law 2) while its quality decreases (law 7). Since their enunciation in the 1970s, the laws have been corroborated on several systems [LRW^{+}97], [EGK^{+}01].

Another indicator of the difficulty of changing systems is the cost of maintenance compared to the global cost of software. Estimates vary between 50% and 90% [Erl00], with a tendency for the most recent estimates to be higher. Erlikh's 90% estimate is incorporated in the recent editions of Sommerville's book on software engineering [Som06].

Software maintenance and evolution is hard because maintainers have to deal with large code bases. This means that a large part of the time involved in maintenance is spent understanding the system. Corbi [Cor89] estimates the portion of time invested in program comprehension to be between 50 and 60 %.

Even with a considerable time spent understanding code, maintenance is not trouble-free. Purushothaman and Perry found that 40% of bugs are introduced while fixing other bugs [PP05], because understanding the complete implications of a change in a large code base is barely possible.

Performing a change is not an easy task either: A simple change can be scattered around the system because of code duplication or because a changing assumption is widely relied upon.

In short, change *is* hard. Maintainers need all the help they can get.

Do they?

In practice, programmers are spending most of their time in static and textual views of a system. Historical information is available in the form of text-based versioning system archives, but is rarely used actively when programming. Thus there is a mismatch between complex evolutionary processes, where software entities are continuously changed, and how maintainers view and interact with software systems. To address this mismatch, evolving systems need to be supported by a better model of evolution itself.

1.2 Reification to The Rescue

Reification is the process of transforming an abstract and immaterial concept, into a concrete and manipulable one. Reification is a powerful tool in software engineering. It is a standard practice in object-oriented design: When designing a software system, a good heuristic is to reify important entities of the problem domain. These entities take a more prominent role in the overall design and are clearly localized in the system.

Reification has also been used successfully to make programming languages more effective by reifying programming language constructs. In general, reifying a construct makes it more expressive, more accessible and altogether more powerful. Some examples are:

- First-class functions passed as arguments to other functions (closures) are used to build higher-level control structures and domain-specific languages. This concept was first found in functional languages.

- The reification of the interpreter in reflective systems [Smi84], or of the object system in an object-oriented language [Mae87], make systems more flexible. Non-functional behavior such as tracing, distribution or debugging can be added to parts or the whole of the system without changing its implementation. A reified interpreter provides hooks to achieve this, while a reflective object-oriented system uses metaclasses.

- Aspect-oriented programming [KLM$^+$97] is a further reification of non-functional concerns as language constructs. Aspects ease the definition and the application of crosscutting concerns to large parts of the system.

- Reifying the call stack in Smalltalk environments was used to implement exception handling and continuations as simple Smalltalk libraries, without modifying the virtual machine or the language itself.

- Osterweil showed that software processes such as testing should be reified [Ost87]. Processes should be described by process descriptions in order to be manipulated and modified by programmers.

In this work, we apply the reification principle to the *changes* performed on a software system. Our goal is to record and make accessible all the changes performed on a system. We name our approach *Change-based Software Evolution*.

We are not the first to consider the evolution of programs as changes. This is a prominent concept in the fields of Software Configuration Management (SCM) and Mining Software Repositories (MSR). These change models have been however incomplete: SCM systems favor versions of text documents for simplicity and genericity. This decision impacts MSR as SCM archives are their primary data sources.

1.3 Change-based Software Evolution

We take a “clean slate” approach to software evolution in order to define a change metamodel freed from the limitations imposed by external circumstances. The change metamodel we introduce has the following characteristics:

- Contrary to SCM systems, it trades generality for semantic awareness, *i.e.,* it deals with the evolution of actual programs and the entities that constitute them, not only lines of text in files.
- It models changes at several granularity levels, from the finest (changes to individual statements) up to the coarsest (aggregating all changes performed during a development session).
- The changes to a system are recorded from an IDE, instead of being recovered from arbitrary snapshots of the program’s source code. The recorded history is more accurate as it does not depend on how often the developer commits or how many versions are selected for study.
- We designed our change metamodel for flexibility. It supports a variety of uses, from analyzing the past evolution of a system, to defining and applying program transformations.

We claim that software evolution can be better supported by reifying the changes programmers make to the system they work on. In this dissertation, we show that an explicit representation of the changes performed on a system helps one to better understand it –reverse engineering– , and then to actually change it –forward engineering.
We formulate our thesis as:

> ***Modeling the evolution of programs with first-class changes improves both their comprehension and their evolution.***

To validate our thesis, we answer the following two research questions:

- *How, and how well, can a change-based model of software evolution assist the* reverse *engineering of a system?*
- *How, and how well, can a change-based model of software evolution assist the* forward *engineering of a system?*

The following section breaks down these research questions in sub-questions, states our contributions and maps them to the overall structure of the document.

1.4 Roadmap

Figure 1.1 shows how the work was performed in the course of this thesis. Research topics are placed in the tree according to their similarity. The chapter in which they are described (if applicable) is indicated. On the right, we indicate the venue in which we published each topic. This thesis is structured in four parts. The first part is the trunk of our work: Based on the shortcomings of evolution models in the literature [RL05], we defined a general model of software evolution emphasizing changes [RL06; RL07a], implemented in a platform named SpyWare [RL08c]. Each branch of the tree represents an area to which we applied change-based software evolution. The branches span spectrum from understanding (part 2) to supporting (part 3) software evolution. The branches covers the topics of reverse engineering and program comprehension [RLL07; Rob07; RL07b], benchmarking for reverse [RPL08] and forward engineering [RL08b; RLP08], and program transformation [RL08a]. Finally, the last part of the dissertation ties these branches together in a unified vision of future work [RL07c].

Part I, First-class Changes: The Why, The What and The How gives the context and explains the concepts of Change-based Software Evolution.

- **Chapter 2**, **Software Evolution Support in Research and Practice**, explores approaches in the domains related to our thesis: SCM, MSR, and IDE monitoring. In the course of this review, we point out limitations of current approaches and extract requirements for our change metamodel.
  Contribution: Requirements for a change-based model of software evolution.

- **Chapter 3**, **Change-based Software Evolution**, presents our change metamodel and the principles which led to its construction. We detail the capabilities of our metamodel and show how it addresses the requirements outlined in Chapter 2.
  Contributions: A change-based model of software evolution satisfying the requirements stated above. An implementation of it for Smalltalk, and a proof of concept for Java.

Part II, How First-class Changes Support System Understanding answers our first research question: How can Change-based Software Evolution assist the reverse engineering of a system? We answer on the levels of reverse engineering, program comprehension and metric definition.

- **Chapter 4**, **Assessing System Evolution**, shows how fine-grained changes can be abstracted to high-level evolutionary facts for the reverse engineering of systems. To support this we introduce a visualization of the change data called the change matrix. Using the change matrix, one can easily locate evolution patterns and extract a high-level evolution scenario of how the system was developed.
  Contributions: A technique supported by an interactive visualization to globally assess the changes performed on parts or the whole of a software system. A catalogue of visual change patterns to characterize the relationships between entities.

- **Chapter 5**, **Characterizing and Understanding Development Sessions**, investigates the use of session-level metrics and session-level visualizations for incremental understanding of sessions. These metrics and visualizations use information which is not recorded by a conventional SCM system. We show how the application of these techniques on fine-grained development session data eases program understanding.
  Contributions: Several metrics and a characterization of development sessions based on change-based information. A process for the incremental understanding of sessions

- **Chapter 6**, **Measuring Evolution: The Case of Coupling**, shows that fine-grained changes increase the accuracy of evolutionary measurements. Logical coupling recovers relationships between entities which might be hidden otherwise. Logical coupling is usually computed at the SCM transaction level. We introduce alternative measures of logical coupling using fine-grained changes, and compare them with the original.
  Contributions: Alternative and more accurate measures of logical coupling, and a benchmark to compare them.

Part III, How First-class Changes Support Software Evolution answers our second research question: How can Change-based Software Evolution assist the forward engineering of systems? We applied Change-based Software Evolution to program transformation and recommender systems.

- **Chapter 7**, **Program Transformation and Evolution**, extends Change-based Software Evolution to support program transformations as change generators. We evaluate how the extension fits in our model, and present a process called example-based program transformation, through which one can record a concrete change and generalize it in a program transformation. Finally, we show that transformations are fully integrated in the system's evolution and discuss the consequences of this.
  Contributions: An extension of our change model to define program transformations. A process to convert concrete recorded changes in generic program transformations.

- **Chapter 8**, **Evaluating Recommendations for Code Completion**, uses the information in our change repository to define a benchmark for a recommender system that is otherwise hard to evaluate, code completion. Based on this benchmark we also define several completion ranking algorithms which are a significant improvement over the state of the art.
  Contributions: A benchmark to evaluate code completion tools. Several algorithms improving completion tools evaluated with the benchmark.

- **Chapter 9**, **Improving Recommendations for Change Prediction**, adopts the same benchmarking strategy for the goal of change prediction. We show that a benchmark based on Change-based Software Evolution is more realistic than one based on SCM data. We implement and evaluate several change prediction algorithms with the help of the benchmark.

Contributions: A benchmark to evaluate change prediction tools. Several algorithms evaluated with the benchmark.

Part IV, First-class Changes: So What? takes a step back from individual validation strategies by considering our techniques as a whole, and concludes the work.

- **Chapter 10**, **Perspectives**, concludes this dissertation by evaluating how well we answered our research questions, discusses our approach and the lessons we learned, and outlines future research directions.

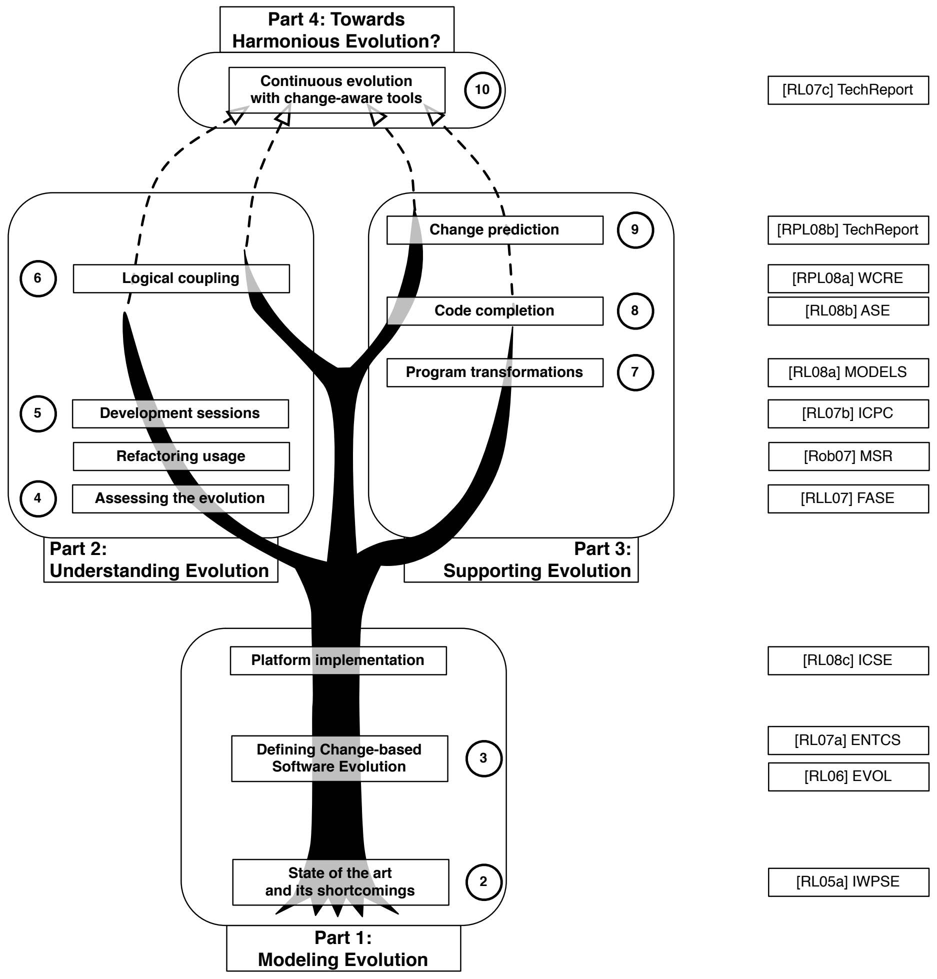


Figure 1.1: Roadmap of our work

Part I

First-class Changes: The Why, The What and The How

Executive Summary

This part of the thesis introduces our central contribution, Change-based Software Evolution. Our goal is to support maintenance and evolution of software systems by modeling the phenomenon of software evolution as it actually happened.

We start in **Chapter 2** *by reviewing the literature in order to compare existing models of software evolution and the maintenance tasks they support. From this review, we infer limitations of each model hindering their support of maintenance tasks. This allows us to draw requirements for a more comprehensive model of software evolution.*

Based on these requirements, we conclude that a unified, clean-slate approach is needed. **Chapter 3** *presents our proposal: Change-based Software Evolution models changes as first-class entities affecting language-specific models of evolving programs. To avoid information loss, we record the changes instead of recovering them*

Chapter 2

Software Evolution Support in Research and Practice

Many approaches have been proposed to address problems related to software evolution. How they model the phenomenon of software evolution has a direct influence on how they can support it. Unfortunately, most approaches model software evolution in an ad-hoc manner. Many reproduce the software evolution model of the SCM system they use as a data source. However the SCM model of software evolution is not adapted to maintenance tasks beyond the ones they directly address, such as versioning and system building.

We review a number of software evolution approaches, and how they model the software evolution phenomenon. In the process, we identify shortcomings in their change model and extract requirements to better support software evolution.

2.1 Introduction

Software evolution has been identified as a source of problems since the 1970s. In nearly 40 years a large amount of research has been performed to ease the changes to evolving systems. We analyze approaches featuring a model of software evolution and list their strengths and shortcomings. From these we extract requirements for a more accurate representation of change in software. The research areas we survey are:

Software Configuration Management (SCM): Although software evolution has only gained wide interest as a research area since the 90's, previous work has been done in SCM. SCM systems had a considerable impact on the practice of software engineering [ELvdH+05]. We review SCM research prototypes and SCM systems used in practice. We outline the characteristics of successful versioning systems and explain them.

Mining Software Repositories (MSR): The field of MSR uses the information contained in software repositories (from SCM systems to mail archives and bug repositories) to analyze their evolution. Applications vary from verifying the laws of software evolution, assisting reverse engineering to building recommender systems. Most approaches based on SCM data reuse their evolution model. We analyze the impact of SCM systems on the kind of research performed in MSR, and find that design decisions beneficial for SCM systems are detrimental to MSR.

Alternatives to SCM and approaches to MSR: More detailed information is available in IDEs, by monitoring programmers while they are interacting with the IDE. We review these approaches and investigate whether and how much they include the concept of change in the data they gather. Finally, we review several approaches which share some of our goals, and use a primarily change-based representation of their data. Most of these approaches are very recent and started while we were working on ours. Some were actually influenced by it. We highlight the differences between these approaches and Change-based Software Evolution.

2.2 Change Representation in SCM

Software Configuration Management is one of the most successful areas of software engineering. The Impact report of Estublier *et al.* [ELvdH+05] gives a thorough account of what characteristics of SCM were successful, or not, and why. In the following we focus on only a few of the many aspects of SCM systems. The characteristics we are interested in are how versioning of resources is performed, and how changes are tracked between versions. Other characteristics such as configuration selection, system building or workspace management are out of our scope. We first list and explain the characteristics we are comparing, before recalling the impact they had on practice, *i.e.,* on the kind of data available for MSR approaches.

2.2.1 How SCM Handles Versioning

There is a slew of approaches to versioning. We refer the interested reader to the survey by Conradi and Westfechtel [CW98] for a comprehensive account of the field. We are more specifically interested in the following dimensions of versioning:

State-based versus change-based versioning: In the state-based model, the versioning system stores the states of the entity, most often in a version tree or graph. Early examples are Rochkind's SCCS [Roc75] and Tichy's RCS [Tic85]. Today, the majority of versioning systems are state-based. To be space-efficient, only one version of a resource (initial of final) can be stored, the other versions being then computed from deltas. In change-based versioning, the changes are stored and the versions are computed from them. Examples are COV by Gulla *et al.* [GKY91] and PIE by Bobrow *et al.* [GB80]. The advantage of change-based versioning is that it allows to easily express change sets, *i.e.,* changes which span more than one resource. Change sets usually have a logical meaning, such as fixing a bug, or implementing a given feature. Although easier to have in a change-based versioning system, change sets are also found in advanced state-based versioning systems. Some systems support both kinds of versioning, such as ICE by Zeller and Snelting [ZS95].

Extensional versus intensional versioning: Using an SCM which features extensional versioning allows one to retrieve any version of the system which was previously committed to the versioning system. Intensional versioning on the other hand allows one to specify and build a version based on a query or configuration rule. A query may also compute on demand a configuration which was not committed to the repository. Intensional versioning is usually implemented in systems based on change sets (the program is composed of a baseline and a combination of change sets), while extensional versioning is the realm of state-based versioning systems, although exceptions do occur. An example is the Adele by Estublier, which supports intensional versioning even if it is state-based [Est95].

General versus domain-specific versioning: A general versioning system is able to version any kind of resource as it does not assume any knowledge about it. In most cases, resources are text files, or binary files. A domain-specific versioning system –such as a programming language aware versioning system– uses the knowledge it has about the domain to handle it with a greater precision. In particular, merging two versions of a resource is much more predictable if the syntax (or even the semantics) of the domain is known. On the other hand, a domain-specific versioning system can only handle its specific domain, and needs to be adapted to be used in another domain. Examples of domain-specific versioning can be found in Perry's Inscape [Per87] and Gandalf by Habermann and Notkin [HN86].

2.2.2 Interaction Models in SCM

Beyond versioning, how people interact with the versioning system is critical. There are several models of interaction with a versioning system:

Checkout/checkin: The checkout/checkin model is the most common interaction model. A developer typically checks out a copy of the files he wants to modify, performs the needed changes, and then checks the files back in. Only then will the versioning system compute the changes or the deltas with respect to the previous version and store them in the repository. Nearly all versioning systems use this model or one of its variants explained below.

Pessimist versus optimist version control: Pessimist and optimist version control are the two main variants of the checkout/checkin model dealing with concurrency issues. Pessimist version control uses locking to prevent more than one user to access a file at the same time. This eliminates conflicts, at the cost of a potentially slower development pace. Optimist version control posits that conflicts are infrequent, and does not restrict the number of people who can access a given file. However, merging algorithms must be implemented to support the occasions in which a conflict actually occurs.

Advanced process support: Advanced SCM systems support other policies beyond optimist and pessimist version control to incorporate changes. For example, the Celine system by Estublier and Garcia [EG06] has flexible policies. One policy is to broadcast changes first to members of the same team, and have a team leader broadcast the changes to the rest of the organization when it is necessary.

Distributed versioning systems: Distributed versioning systems do not rely on a central repository. Getting a snapshot of the source code also involves getting a local copy of the repository which subsequent commits will be stored into. This makes branching easy. When branches are merged in a central repository, the history can be brought back as well if needed. Distributed versioning is quickly gaining supporters among open-source projects. Example systems are git [1], darcs [2] and mercurial [3].

Operation recording: All of the interaction models described so far are variants of the checkout/checkin model. Few approaches really diverge from it. The alternative is to record the changes performed in an environment, rather than inferring them at commit time. Such an approach was employed in the CAMERA system by Lippe and van Oosterom, which recorded operations to implement an advanced merging algorithm producing better results [LvO92].

[1]http://git.or.cz
[2]http://darcs.net
[3]http://www.selenic.com/mercurial

2.2.3 The State of the Practice in SCM

So what makes an SCM system successful? The Impact report on SCM states it plainly:

> "Of note is that virtually every SCM system is carefully designed to be independent from any programming language or application semantics. [...] We believe this is a strong contributor to the success of SCM systems."[ELvdH+05]

The majority of SCM systems in use today are general-purpose, file-based SCM systems relying on the optimist checkout/checkin interaction model. The most advanced versioning systems have a degree of changeset support built on top of state-based versioning, but do not fully use change-based versioning.

This is not surprising: A typical project needs to version a large number of entities of different types, from source code files to documentation in various formats (web pages, PDF manuals, READMEs), build files (Makefiles), or binary data (images, *etc.*). A project may be implemented in several languages. This renders language-specific versioning not really usable for most projects. One could conceive using two versioning systems, but this incurs too much overhead.

In practice, people are willing to compromise on merging capabilities in order to keep using a generic versioning system. This is also a reason why checkout/checkin systems are still used: If operation recording offers only advantages when merging, it is not worth switching versioning systems and giving up the support for other file types.

Inertia is another factor. Changing from a versioning system to another implies learning a new tool, so the benefit needs to be substantial. Switching during the life of a project is even riskier, as the data in the old repository is valuable. If no repository conversion tool exists, the data risks being lost or forgotten.

If we analyze the versioning systems used in the open-source world, we see these forces in action. A few years ago, CVS was the dominant versioning system, with barely any competition. In a survey of versioning systems [RL05] we predicted that open-source software developers would switch to Subversion. Today, Subversion is the dominant open-source versioning system for several reasons. It is a significant improvement over CVS: It versions both files and directories, whereas CVS versions only files, and features some support for changesets as it has transactions. Yet, it remains very close to CVS, as the commands are very similar. Its stated goal was to be an incremental improvement over CVS. Finally, automated support exist to convert a CVS repository to a Subversion repository.

Distributed versioning systems are increasingly popular: Git hosts Linux (which is not surprising since the same person is behind both projects). Distributed versioning is a significant improvement as it makes branching a system much easier, which is a key point for open-source software. However, these versioning systems still keep their language-independent design and follow the checkout/checkin model.

2.3 Impact of SCM Practice on the Research of MSR

The Impact report on SCM states that:

> “A side effect of the popularity and long-term use of SCM systems has been the recent discovery that they serve as an excellent source of data [...]. A new field, mining software repositories, has sprung up [...]. Without SCM systems, this entire field would not be in existence today.” [ELvdH$^+$05]

SCM has indeed caused the existence of the MSR field. Their goals are however not the same. As a consequence, the design decisions taken by SCM systems which contributed to their success are obstacles for MSR research. Since no other information source is available, MSR research must adapt to the versioning systems which are widely used in practice. Today, these are CVS and Subversion.

Two particular SCM design decisions, language-independence and the checkout/checkin model, cause a significant part of MSR research to be either focused on reconstructing the evolution of software, or on making high-level observations about it. Gîrba’s metamodel Hismo [Gîr05] is the most advanced formalization of version-based evolution models. It addresses the first shortcoming to some extent, but not the second, and is still sensible to their interplay.

2.3.1 The shortcomings of SCM for MSR

Versioning files increases data processing Being language-independent makes SCM systems versatile: They can version any kind of file, even binaries. This automatically makes any detailed analysis of a software system version much harder, as each system version must be parsed, which is an expensive process. Without parsing the system, only high-level analyses such as the evolution of the number of lines of code or the number of files in the system are possible.

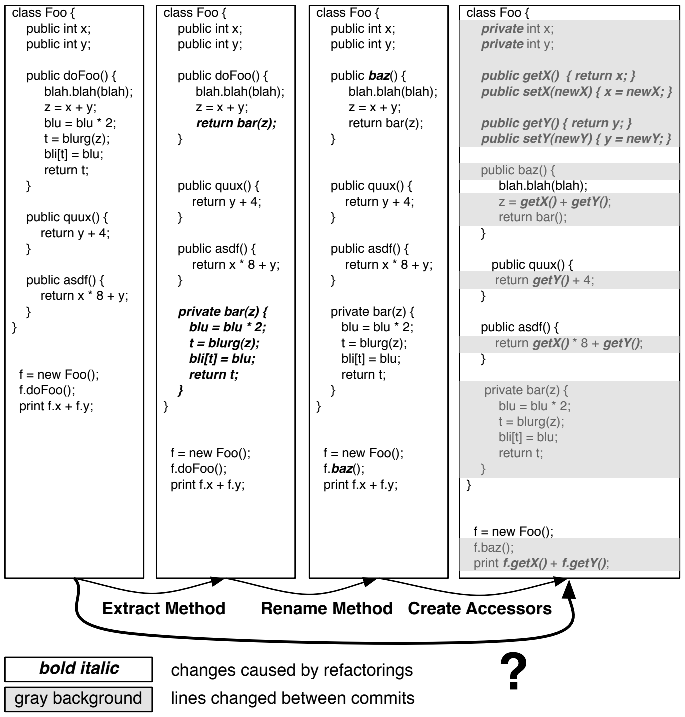
Since several versions of the system must be considered, the problem of traceability arises. Entities must be matched across multiple versions of the system. The usual heuristic to consider that entities with the same name are the same, does not cover all the cases. Each entity could have been renamed, or could have moved from one place in the system to another. Without a good matching algorithm, spurious additions and deletions of entities will be recorded. Careful and costly examination of two successive versions of the system is needed to map an entity to its sibling in the next version: Multi-version entity matching, or origin analysis, is still an active research area.

Parsing and multi-version matching may be costly, but they allow one to analyze the evolution of systems with more precision than by using only files and lines. For instance, Gîrba *et al.* examined the evolution of class hierarchies [GLD05]. Zimmermann *et al.* used lightweight parsing of the entities added or deleted in a transaction for change prediction [ZWDZ04], while Dagenais *et al.* used it to recommend changes when a framework evolves [DR08].

Taking snapshots loses data The second problem lies with the interaction model of major SCM systems, the checkout/checkin or any of its variants. In this model, a developer interacts with the SCM system only when he wants to update his working copy or when he commits his changes to the repository. This is the only time in which the SCM system can determine the changes the developer made to the system.

However, there are no guarantees of how often developers commit. An arbitrary amount of change can have taken place before a commit. If only a few changes are committed at the same time, it is still easy to differentiate between them. On the other hand, if more changes are committed at the same time, then inferring what each change does becomes a problem. While the design choice of being language-independent is merely an inconvenience incurring extra preprocessing of the data, the checkout/checkin model makes SCM systems actually *lose information*.

In addition the checkout/checkin model does not record the exact sequence of changes performed in a commit. All changes in a transaction will have the same time-stamp. Their order can *not* be inferred.



```
class Foo {
    public int x;
    public int y;

    public doFoo() {
        blah.blah(blah);
        z = x + y;
        blu = blu * 2;
        t = blurg(z);
        bli[t] = blu;
        return t;
    }

    public quux() {
        return y + 4;
    }

    public asdf() {
        return x * 8 + y;
    }
}

f = new Foo();
f.doFoo();
print f.x + f.y;
```

```
class Foo {
    public int x;
    public int y;

    public doFoo() {
        blah.blah(blah);
        z = x + y;
        return bar(z);
    }

    public quux() {
        return y + 4;
    }

    public asdf() {
        return x * 8 + y;
    }

    private bar(z) {
        blu = blu * 2;
        t = blurg(z);
        bli[t] = blu;
        return t;
    }
}

f = new Foo();
f.doFoo();
print f.x + f.y;
```

```
class Foo {
    public int x;
    public int y;

    public baz() {
        blah.blah(blah);
        z = x + y;
        return bar(z);
    }

    public quux() {
        return y + 4;
    }

    public asdf() {
        return x * 8 + y;
    }

    private bar(z) {
        blu = blu * 2;
        t = blurg(z);
        bli[t] = blu;
        return t;
    }
}

f = new Foo();
f.baz();
print f.x + f.y;
```

```
class Foo {
    private int x;
    private int y;

    public getX()  { return x; }
    public setX(newX) { x = newX; }

    public getY() { return y; }
    public setY(newY) { y = newY; }

    public baz() {
        blah.blah(blah);
        z = getX() + getY();
        return bar();
    }

    public quux() {
        return getY() + 4;
    }

    public asdf() {
        return getX() * 8 + getY();
    }

    private bar(z) {
        blu = blu * 2;
        t = blurg(z);
        bli[t] = blu;
        return t;
    }
}

f = new Foo();
f.baz();
print f.getX() + f.getY();
```

Figure 2.1: Simple refactoring scenario leading to evolution information loss.

Example: Meet Alice One might think these two shortcomings are not too much a problem, especially since only one of them involves information loss. Figure 2.1 shows how this loss of information can significantly degrade the knowledge inferred about a system. In this simple scenario, Alice, a developer, starts a short refactoring session, in which she refactors the method `doFoo`. She:

- applies the "Extract Method" refactoring to `doFoo`: This extracts a block of statements she selects in a new method `bar`;
- applies "Create Accessors" to attributes `x` and `y`. The refactoring replaces direct accesses to instance variables `x` and `y` with accessors throughout the entire system;
- applies "Rename Method" to `doFoo`. `doFoo` is renamed to `baz`, replacing all references to `doFoo` in the code base.

Alice then commits these changes. This is a very small commit, less than a minute of work, since all these refactoring operations can be semi-automated: In current IDEs, they are only a right-click away. 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 could 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 renaming 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[Fow02]– have been performed. There were no logical changes to the system, yet this commit caused many physical changes: Its importance measured in lines of code is overestimated. CVS would report that 11 lines were removed, and 18 lines were added. Extra processing is needed to make that figure accurate.

The simple scenario depicted above assumes that a developer commits after every couple of minutes of work. Table 2.1 presents statistics gathered on 16 open-source projects using the Subversion version control system. All the commits were grouped by author and by date. The next to last column shows that an average developer will perform more than one commit per day barely 15% of the time. A developer such as Alice would on the other hand perform dozens of commits daily. When two or more commits are performed on the same day, the average distance between them is nearly four hours, far more than the five minutes taken above (We used a sliding time window of 8 hours to determine whether two commits took place on the same day). Finally, a quick look at the distribution of commits by size shows that

Project name	Number of commits	% 1 file	% 2-4 files	% 5-9 files	% 10+ files	% days with 2+ commits	interval (minutes)
Ant	14,078	96.25	3.35	0.27	0.13	17.35	227
Django	4,812	87.43	12.57	0	0	10.83	232
Gcc	87,900	47.26	40.64	6.64	5.46	24.72	209
Gimp	23,215	91.68	7.85	0.33	0.13	21.39	235
Glib	5,684	88.79	10.66	0.44	0.11	32.12	181
Gnome-desktop	4,195	89.92	9.58	0.38	0.12	41.12	178
Gnome-utils	6,611	80.34	19.32	0.33	0.02	29.53	183
Httpd	39,801	56.17	40.76	2.89	0.17	19.96	209
Inkscape	14,519	90.92	8.83	0.22	0.03	17.65	228
Jakarta	70,654	77.43	20.74	1.64	0.18	17.04	217
Jboss	5,962	95.67	4.29	0.03	0	19.52	220
KDE	817,795	78.48	20.59	0.83	0.10	13.05	231
Lucene	14,078	80.52	18.45	0.93	0.10	17.35	227
Ruby on Rails	9,251	96.25	3.35	0.27	0.13	12.88	240
Spamassassin	10,270	91.17	8.26	0.50	0.08	17.58	222
Subversion	21,729	50.26	47.83	1.70	0.21	25.70	222
Total	1,158,824	75.69	22.41	1.38	0.52	15.16	226

Table 2.1: Per-author commit frequency in several open-source projects

if 75% of them change a single file, 25% change a larger number of files. This is particularly problematic for the 2% of commits which span changes across more than 5 files, indicating either large changes or crosscutting.

Another factor at play when analyzing open-source repositories is the patch practice. A core group of developers are free to commit to the central repository, but most people do not have access to it. If they want to submit a change to the system, they will submit a patch file (essentially a delta between their version and the standard version). The patch will be reviewed by some of the core committers, and if deemed satisfactory, committed to the central repository. This means that features are proposed to the core team when they are stable: The evolution which led to the feature implementation happened outside of the repository and is hence lost.

When 1 + 1 = 3 Finally, the conjunction of both shortcomings yields further problems. To dampen the checkout/checkin problem, one would want to have as many SCM commits as possible, in order to get a more accurate vision of the evolution of the system. In essence, one would want to analyze as many versions as possible to get the smallest differences between each version.

This however directly conflicts with the first shortcoming. Since fully parsing an entire system is an expensive operation, parsing 10,000 versions of one system is even more so. This is why most software evolution analyses use sampling, and select only a few versions of the system they study, typically under a dozen. Sampling is so common, that Kenyon by Bevan *et al.* [BEJWKG05], a tool platform aimed at easing software evolution analyses by automating common tasks, listed sampling as a requirement for the tool.

In short, even if the developers of the system are disciplined enough to commit early and often to the SCM system in order to minimize differences between versions, the sheer number of versions forces evolution researchers to only select a few of them. The farther apart two versions are, the more changes between them, and the more difficult it becomes to tell individual changes apart. Selecting 10 versions out of five years of history leaves one version every six months, a far cry from the two minutes scenario we used as an example. Entire parts of the system seem to appear at once with no history whatsoever, essentially defeating the purpose of evolution analysis. How can one pinpoints shortcomings of a system based on its history if there is no history to be found?

Conclusion Given the shortcomings of SCMs as an accurate evolutionary source and the considerable data loss they incur, it is not surprising that among the currents of MSR research, two of the main ones are high-level analysis, and evolution reconstruction. In the following sections, we review high-level approaches, contrast them with full-model approaches, and then review evolution reconstruction solutions.

2.3.2 High-level evolution analysis

High-level analysis considers that it is too costly to parse the system and hence uses information which is more easily accessible such as commit logs, number of lines of code and the number of files in a system. A commit log stores for each commit its author, its date, and the files modified during the commit. Transactions have to be reconstructed with CVS. SVN does not mention in the commit log the number of lines added and deleted for each file.

Logical coupling introduced by Gall *et al.* is a high-level solution to the coupling problem [GHJ98]. Instead of detecting which entity depends on which other by analyzing method calls between them, logical coupling counts the number of times two entities changed together.

Robles, Herraiz *et al.* showed that simply counting the number of lines of code of modules evolving over time can give some insights about the evolution of systems [RAGBH05], [HGBR08]. Godfrey and Tu found that some open-source systems such as Linux have a superlinear growth [GT00], instead of the expected linear one. A finer analysis can consider, beyond lines of code, the physical structure of the system as files and directories, as done by Capiluppi *et al.* [CMR04].

Authorship patterns in evolutionary files has been analyzed through the ownership maps of systems by Gîrba *et al.* [GKSD05] and fractal figures by D'Ambros [DLG05]. The former emphasizes the time dimension, the latter the structure of the system. Other sources of data are considered. Fischer *et al.* linked version control and bug tracking information [FPG03], which was visualized by D'Ambros and Lanza [DL06b]. Recently, Bird *et al.* analyzed mailing list archives [BGD$^+$06].

Conclusion If a large number of versions can be considered when performing high-level analysis, its insights are limited. In addition, their accuracy has been questioned by Chen *et al.* [CSY$^+$04] who expressed doubts about the accuracy of commit logs, when they compared them with the actual changes found in the files.

2.3.3 Full model evolution analysis

Full model evolution analysis is more coarse-grained in terms of number of versions, but yields more precise results. Analyzing the evolution of more complete program models, researchers were able to identify more precise characteristics or shortcomings of systems. Among the numerous approaches that have been tried, we mention a few.

Holt and Pak visualized the evolution of the architecture of a system across two versions [HP96]. Xing and Stroulia [XS05] focus on detecting evolutionary phases of classes, such as rapidly developing, intense evolution, slowly developing, steady-state, and restructuring.

Gîrba formalized the evolution of systems for which the SCM data is available in his Hismo metamodel [Gîr05]. Based on Hismo, Gîrba *et al.* analyzed the evolution of entire class hierarchies [GLD05], while Lungu *et al.* analyzed the relationships between packages [LLG06], and subsequently the evolution of their relationships [LL07]. Wettel and Lanza analyzed the evolution of systems at the system level, while also taking into account the evolution of classes and methods [WL08]. Raţiu *et al.* defined and evaluated the concept of history-based detection strategies, which differentiates between stable and unstable defects [RDGM04].

Conclusion Fuller analyses permit deeper insights about the evolution of language-level entities in the system. However the number of versions analyzed is usually limited. For instance, Holt and Pak [HP96] considers two versions at a time. Xing and Stroulia [XS05] analyses 31 versions in 4 years, which amounts to less than 1 per month. Raţiu *et al.* [RDGM04] analyses 40 versions out of the 600 available on a 10 year period. This means that the history available is significantly reduced, in turn reducing the accuracy of the approaches.

2.3.4 Evolution reconstruction approaches

Evolution reconstruction tries to make up for the lost information by inferring the changes that happened during the evolution.

Refactoring detection According to Dig and Johnson, refactorings are a significant portion (80%) of API-breaking changes [DJ05]. It is possible to automatically update code which was broken by a refactoring, as demonstrated by Henkel and Diwan [HD05] or by Savga *et al.* [SRG08], provided they are recorded (from the IDE) or detected (from MSR archives).

Weißgerber and Diehl present an approach to detect refactorings which were performed between two versions of a system [WD06]. So do Dig *et al.* [DCMJ06] and Taneja *et al.*

[TDX07]. Earlier, Demeyer *et al.* used metrics [DDN00]. These approaches however detect only a subset of refactorings, mainly "Rename" and "Move" refactorings.

Version matching Matching entities across versions is a well-known problem, since entities can be renamed or moved between two versions. It is however essential if one wants to analyze the entire history of a given entity. Without it, the entity's history will be split, with one entity disappearing while the other appears.

Tu and Godfrey [TG02] use origin analysis to determine if an entity is effectively the same in several versions of a system. The approach was refined in [GZ05], to detect entities being merged or split with another. The problem of renamed functions was also tackled by Kim, Pan and Whitehead [KPEJW05]. Kim, Notkin and Grossman propose another approach [KNG07] using change rules and an inference algorithm.

Clone detection Detecting duplicated code and showing how it evolves is a relevant problem. Duplicated code poses a maintenance problem, since a change to one clone usually implies changing all the other clones to avoid bugs. Detecting clones across versions allows one to see which clone instance is the originator, and see the evolution of a clone group across time, as shown by Adar and Kim [AK07]. Since clone detection is resource intensive, a small amount of work has been performed in this area.

Contradictory claims have been made about the harmfulness of clones. The conventional wisdom is that clones should be avoided: When a clone group is found, it should be refactored to remove the duplication by abstracting away the common behavior. However, recent work by Kim *et al.* [KSNM05], or Kapser and Godfrey [KG06] suggest that this is not always the best course of action. Some clones are better left alone, as they are too hard to refactor, or are going to evolve differently. To handle that situation, Toomin *et al.* proposed linked editing [TBG04], while Duala-Ekoko and Robillard presented a clone tracking tool [DER07]

Line-based evolution At an even lower level, an approach by Canfora *et al.* is dedicated to differentiate between lines added, deleted and simply changed in a CVS commit [CCP07]. By itself, the only information CVS gives is the number of lines added and deleted. Even a single character change would be interpreted as the addition of one line and the removal of another. Of note, Subversion does not provide any estimation of the number of lines added and deleted in a transaction: This has to be computed separately.

Conclusion A lot of approaches exist to recover a system's evolution with more accuracy. All of them are limited by the change amount between versions. They are all time-consuming, strengthening the problem of limited versions. To date, these techniques have been used in isolation rather than being combined.

2.4 Alternative Approaches

2.4.1 IDE monitoring as an Alternative to SCM Archives

In recent years, a sizable proportion of programmers have begun to use Integrated Development Environments (IDEs)[LW07]. Modern IDEs are also very flexible and feature a plug-in architecture third-parties can build on.

For these reasons, Eclipse, the most used Java IDE, is frequently adopted by the research community as a platform to implement research prototypes. A review of these shows that by using IDEs, one can get around the limitations of SCMs, by getting some development information during the time where the SCM is not solicited. This is possible since an IDE such as Eclipse features an event notification mechanism to which interested parties can suscribe.

Context-building tools Mylyn (formerly Mylar) by Kersten and Murphy [KM05; KM06] determines what entities are interesting to a developer based on his recent interactions. It uses a degree-of-interest (DOI) model in which entities which are browsed or edited see their degree of interest increased, while it otherwise slowly fades with time. Mylyn tracks navigation and editions in the IDE at a shallow level: It tracks which entity was changed, but not how or to what extent.

NavTracks by Singer *et al.* employs a similar approach [SES05], but focuses on the navigation in files, proposing files which are likely to be navigated to next. TeamTracks by DeLine *et al.* [DCR05] features a similar name and approach: It displays a filtered view of entities based on the entity in focus. Finally, Parnin and Görg propose another similar approach were they reify usage contexts [PG06].

Interaction Coupling Zou *et al.* propose an alternative to logical coupling, called interaction coupling, which takes into account both the changes to the program and the navigation [ZGH07]. In particular, it needs less data (*i.e.,* a shorter history) than SCM-based logical coupling before returning results.

Awareness Awareness tools, which tell developers when they are working on the same part of the system, can be implemented using a finer-grained IDE monitoring, as shown by Schümmer and Haake [SH01] (at the method level), instead of the more widespread monitoring of files taken by Estublier and Garcia [EG05] or Sarma *et al.* [SNvdH03].

Conclusion If these approaches use a finer type of information, none so far feature a deep analysis of the entities they monitor, such as detecting the kind of change that was applied to it. Parnin *et al.* proposed to combine traditional MSR with IDE data [PGR06], so that interactions are also considered. However, this does not help in finding more precise changes.

2.4.2 Change-based approaches

The approaches we saw above have some kind of change representation which is either based on version in the case of SCM and MSR, or very shallow in the case of IDE monitoring (presence or absence of changes). Here we review more complete change representations which are similar to ours.

Change-based and refactoring-aware versioning systems Smalltalk has featured a change model for some time, in the form of change sets. This model however is limited since only changes to methods and classes are described. This model has been extended to build a fuller SCM system named PIE by Goldstein and Bobrow [GB80], in which features of the system are each represented as a distinct layer. The closest approach to ours is operation-based merging by Lippe and van Oosterom as used in the CAMERA system [LvO92], where operations are recorded and manipulated to perform the merging of conflicting edits. However, the operations are not explicitly specified as the paper describes operation-based merging from a generic standpoint. Operation-based merging focuses only on the merging problem, as part of collaborative development [Lip92]. The approach has been extended by Freese [Fre07], with the objective to also include refactoring-aware versioning. However the approach considers only the merging problem. Another similar approach is taken by Kögel [KÖ8]. First-class changes are used to version UML models. This representation is natural since UML models are not text-based. Kögel employs a change hierarchy similar to ours at the lowest levels, but is interested mainly in versioning.

Several versioning systems are change-based, but still remain language-independent, and as such keep much of the same problems: Translating first-class changes to lines in AST-level changes is not trivial. These systems are also snapshot-based systems. Those are too numerous to list here. A recent and interesting system is the patch-based Darcs[4], where every change is stored as a patch, and a theory of patches and the operation they support is provided.

Several versioning systems support explicit refactoring as a kind of change. Ekman and Asklund's system [EA04] stores ASTs of entities, and separates edit operations from refactorings. Dig *et al.* present a system [DMJN07] based on Molhado [NMBT05], a flexible SCM infrastructure by N'Guyen *et al.* MolhadoRef separates edits which are versioned normally, from refactorings, which are stored separately.

Accurate evolution reconstruction A few approaches use versioning system archives to build a detailed change representation.

ChangeDistilling by Fluri *et al.* [FWPG07] parses source code files and uses AST differencing to build a more accurate change representation. The AST they use goes down to the control flow level: Instructions such as iterations and loops are modeled, but individual

[4]http://en.wikibooks.org/wiki/Understanding_darcs/Patch_theory

statements are strings only. Change Distilling has been used for software evolution analysis, including a change classification [FG06] by their significance.

Schneider *et al.* mined the local edit history they recorded [SGPP04]. Their tool, ProjectWatcher, uses a "shadow repository" where they commit changes automatically, thus not relying on the developer to commit. A fact extractor is then used to infer relations between entities in Java, such as classes, packages, methods and calls. Not everything is parsed. The system was primarily used for awareness visualization.

Change-based models Finally, several models feature change representations similar to ours, or similar tactics to record them.

Blanc *et al.* [BMMM08] encode models as a sequence of construction operation (changes) to detect inconsistencies in them. They however do not record or use any history.

Changeboxes by Zumkher *et al.* [Zum07; DGL$^+$07] model changes as a first-class entity with the goal to make several versions of a system coexist at runtime. It also features basic SCM capabilities, such as merging. The change representation models entities up to the method level, but not below it.

Cheops [EVC$^+$07] is another model of first-class changes aimed at run-time evolution of systems. The authors took an early version of our change model as an example and extended it. They also use the FAMIX model as their program model [TDD00], while we use our own program model which is simpler.

Omori and Maruyama implemented a tool named OperationRecorder for Eclipse [OM08]. Their approach is directly inspired by ours, but features a different change recording approach.

Chan *et al.* also record changes as they happen from Eclipse [CCB07]. However they adopted a language-independent approach, trading accuracy in analyses for genericity. They propose several visualizations of the change data.

Conclusion. Over time, several approaches explicitly modeling software change have been proposed; their number have increased recently. The domains of application are quite specific and vary from versioning systems (targeting merging, collaboration and domain-specific areas such as MDE), to evolution reconstruction approaches aiming for accuracy, and runtime evolution of systems.

2.5 Summary

Versioning systems have to cover a variety of tasks, such as workspace management, policies, system configuration and building, beyond mere versioning. So far, successful versioning systems have been language-independent and non-intrusive. This led them to version files according to the checkout/checkin version model.

MSR approaches depend on the versioning system to gather evolutionary data. They hence rely on general change models which do not provide many insights about the evolution of systems, beyond high level observations. Post-processing of the data is possible to parse successive versions of the system, but is expensive. There is thus a trade-off between the number of versions considered *i.e.,* the accuracy of the history, and the accuracy of the system's model. The more precise the system model is, the larger the time periods between two successive versions.

IDE monitoring tools bypass or complement the information found in SCM repositories with IDE usage information obtained by tracking what the programmer is doing. So far, the change models used in IDE monitoring (when one was used), have been shallow: One knows that a program entity was changed, but now how or by how much. Other approaches only use the navigation information, where by definition there is no change model whatsoever.

Change-based approaches are few and recent for the most part. Several models have been proposed. Some only model refactorings, and use classical versioning for other edits. Some infer changes from CVS archives, while other record them. The granularity varies: Some stay language independent, other model several kinds of entities. Some of them model entire ASTs, others parts of it, and others stop at the method level. None model all changes while also adapting to the language and recording the changes from the IDE.

Conclusions From our literature review, we extract the following conclusions:

- SCM systems are an inadequate source of information if one wants to build an accurate model of software evolution.
- MSR has found a variety of uses, from reverse engineering to change prediction, to analyzing clone evolution and refactoring detection. Our change-based model of software evolution should support a variety of activities, from high-level ones to lower-level ones.
- Precise approaches such as refactoring detection, change prediction, generally rely on at least some knowledge of the language being used, while reverse engineering rely on a fuller knowledge of it. Hence supporting language-level entities is critical.
- IDEs allow one to gather very precise information about the way programmers use the IDE. The open architecture of IDEs allows one to be notified of what developers do fairly easily. So far, the use of this information has been limited. We believe much more can be achieved with more detailed IDE monitoring.

Chapter 3

Change-Based Software Evolution

At the heart of the software life cycle is change. We established that to better support change, we need an accurate model of it. We present our change-based model of software evolution and explain how it addresses some of the shortcomings of other approaches. Our model is based on the following principles:

- *Programs need to be represented accurately: A program state is represented by an Abstract Syntax Tree of the entities composing it.*
- *Changes need to be represented accurately: A program's history is a sequence of changes. Each change, when executed, produces a program state in the form of an AST. Changes can be composed to form higher-level changes.*
- *Changes should be recorded, not recovered: To achieve a greater accuracy, changes are recorded in an IDE as they happen, rather than being recovered from versioning system archives.*

3.1 Introduction

This chapter details our model of change-based software evolution. From our literature review we identified strengths and shortcomings of state of the art approaches. From these we extracted high-level guidelines, or principles, that support our approach. We first list and justify each of the principles behind our approach, before describing our change meta-model, our program model and our change recording strategies. Finally, we outline the validation steps we took.

3.2 Principles of Change-based Software Evolution

Principle 1: Programs instead of Text

Systems use the finest possible representation, abstract syntax trees.

If we wish to model and analyze evolution accurately, we need to adopt the most accurate data representation we can. The state of a program is most accurately described as an Abstract Syntax Tree (AST). We model the structure of the system as a tree of entities (at both coarse and fine levels), and the references between entities such as accesses to variables, calls to methods, *etc.*

Pros:

- We build an accurate representation from the ground up: We have seen that multiple analyses are performed to assist both reverse and forward engineering. If some are lightweight (like file-level change coupling), others require either shallow parsing (method level change prediction and coupling), or full parsing (class hierarchy evolution analysis). According to the saying, if one can do the most, he can also do the least: A fully parsed solution contains the information needed for less detailed analyses. For instance, it is always possible to generate source code if counting the lines of code is needed.

- To perform accurate analyses, an accurate representation is needed. We also know that parsing and matching entities is expensive. It seems more economical to perform it only once and have a direct representation that can be accessed from then on.

- ASTs are insensible to layout modifications. A class of low-significance changes can be filtered out without needing a special analysis. Other kinds of changes are detected more easily.

Cons:

- Lightweight representations would be less memory intensive. Maintaining a full system AST occupies more memory than a simpler model encoding only file names and number of lines. The scalability of our approach could be an issue for large systems. We think however that the amount of memory available in today's –and tomorrow's– machines makes it usable. As we show with the second principle, we do not maintain AST representations of every versions of the system at every time: The ASTs are computed on demand.

- Parsing is language dependent. We need at least a parser for the given language: If none is available, a substantial effort will be needed to build one. However, without such a parser, no advanced analysis would be possible anyways.

Principle 2: Changes instead of versions

Changes are represented as first-class entities –as executable AST operations supporting composition.

We want to model the phenomenon of change itself. If our base representation is the AST of a program, it follows that changes are AST operations, hence simple tree operations. We also need a composition mechanism to support higher-level changes, such as changes touching several parts of the tree. An example is refactoring [Fow02]: A refactoring such as "Rename Method" actually changes several methods since it has to update all the references to the renamed method. Since there are many types of changes that can occur in a system, each with different mechanisms, our change model needs to be flexible enough to accommodate them all.

If changes at the low level are simple tree operations, they can be made *executable* and can then produce ASTs. If a mapping exists between a change and its opposite, each change can be *undone*. These two properties can also be transmitted to higher-level changes.

Pros:

- First-class changes are more accurate than versions. The only way to encode that a refactoring occurred between two versions is to state it outside of the version model. First-class changes do just that, except that they model *every* change that happened between two *arbitrary* versions of the system.

- First-class changes are a superset of versions. Since they can be executed and undone, they can *produce* a version of the system as an AST if this is needed. One can see first-class changes as deltas used behind the scenes by most versioning systems, with the difference that deltas must work with every kind of file, and are as such either text-based, or binary.

- Changes use less memory than versions. Accurate approaches to evolution analysis model a system as successive versions. The default approach is to have a copy of each entity for each version, even if it has not changed between these two versions. A more space-efficient scheme could of course be implemented (such as deltas in SCM), but a change-based implementation provides it "for free". One could add that the more space-efficient this encoding scheme is, the more similar to an accurate change representation it becomes.

Cons:

- One could argue that executing changes to produce versions may not be scalable. Beyond a certain size, it would become intractable. Initial evidence for medium-sized projects –such as our own prototype, which ranges in the tens of thousands of lines of code during the course of 3 implementation years– shows that we have not reached that point yet. Replaying the entire history of a system is in the order of minutes. Optimizations are of course possible: Storing snapshots of the system at several points in time to have a hybrid between changes and versions is an approach we have not investigated yet. We have on the other hand experimented with a scheme to access quickly given entities by selecting only necessary changes (Section 3.4.4). This makes accessing the state of any entity at any time a matter of seconds.

Principle 3: Record instead of recover

Changes are recorded from the IDE –instead of being reconstructed from SCM archives.

Our review of the evolution reconstruction research in MSR convinced us to look for another approach. We want to avoid the trade-off between the number of versions one can consider and the depth of the analysis one can perform on them. IDE monitoring gives access to a large amount of information that the checkout/checkin interaction model of SCM system loses. Therefore, we decided to *record* changes *as they happen* in the IDE, rather than recovering them.

Pros:

- Recording is simpler than reconstructing. Whenever our system is informed of a change, it can query the IDE for more information about it, in order to build our change representation. This amounts to perform a difference between two versions of a program, but with two advantages: (1) the difference is as small as possible since we are notified of changes immediately (2) we know which part of the system just changed, so the differencing algorithm is used on less data, and entity matching is simplified.

- Recording gives us more information. When we are notified of a change, one of the simplest query we can make is to ask for the time stamp. This allows us to give a timestamp to each change with a precision up to the second. In essence, we can record the entire working session which resulted in a commit, rather than only reconstructing its outcome.

- IDE integration is anyway necessary. Tool implementation are more and more released as IDE extensions. If we want to produce tools that assist a programmer, it is only natural to also use the IDE to record the changes. In the last two Future of Software Engineering conferences (co-located with ICSE), invited papers in reverse engineering by Müller *et al.*, and by Canfora and Di Penta, evoked the vision of "continuous reverse engineering", where developers themselves interleave forward and reverse engineering in their day-to-day activities [MJS$^+$00; CP07]. Continuous reverse engineering requires easy access to the reverse engineering tools while programming.

Cons:

- Our approach requires the programmer to use an IDE. Programmers using a classical text editor are left in the cold. However, we believe that in a few years the overwhelming majority of programmers will be using IDEs for all their daily tasks. Most students today learn to program using IDEs and prefer them over classical text editors [LW07].

- Our approach is IDE-specific. Since we rely on IDE monitoring, at least one part of our approach has to be reimplemented every time we adapt it to a new IDE. However, the problem would be still be valid since we would need to build tools as IDE plugins anyway.

- What if some changes are performed outside of the IDE? Sometimes, programmers do quick changes outside of the IDE, which would not be recorded. These cases are however a small minority of all edits: Any long programming task is much more comfortable if done in an IDE. In such cases, evolution reconstruction approaches such as ChangeDistilling by Fluri *et al.* [FWPG07] could be employed to import those changes in the model. Of course, those would appear as a "clump" of changes –as they would not have a precise timestamp–, but this still would give us a reasonable approximation of the evolution, under the assumption that these changes are small.

3.3 Program Representation

Our first principle is that we should adopt a domain-specific representation of programs. It should however be easy to define a new problem domain and adapt our approach to it. This section describes our program representation and how it adapts to particular programming languages.

3.3.1 Abstract Syntax Tree Format

Generic AST representation First-class changes are applied to programs. Our first task is to define an adequate representation of a program in our model. Our program representation has the following goals:

Simplicity: The program representation is not the primary focus of interest – the changes are.

Genericity: Our program representation will contain language-dependent data. It should however be as language-independent as possible. The program model should be adaptable with minimal effort to other languages, and support a variety of analyses.

Flexibility: If an extension is needed for a programming language or a new kind of analysis, then it should require minimal effort to add it.

Fine-grained: For maximum accuracy, we want to model entities up to the statement level.

With these constraints at hand, we decided to define our own program model instead of adopting a program model which was already defined. FAMIX, by Tichelaar *et al.* [TDD00] was considered. FAMIX does however not model the entire AST of a method, only the invocation of messages and accesses to variables in it, a decision reasonable for a model geared towards reverse engineering. The author of FAMIX furthermore stated that UML is not adequate for reverse engineering without extensions, and chose to use FAMIX rather than extend UML [DDT99]. Both models would also require extensive effort to be implemented, failing the simplicity constraint.

Since our system needs to support several types of analysis, we opted for the simplest AST representation possible. Each of our AST nodes can be described by Figure 3.1. The attributes of the AST nodes are detailed below:

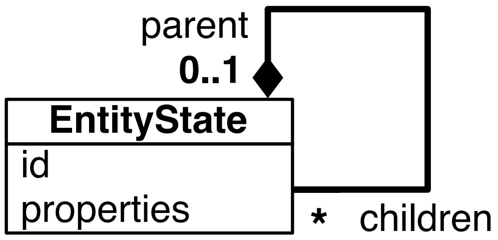


Figure 3.1: A node of the program's AST

id: Each AST node has a unique identifier to unequivocally identify it.

parent: The parent of an entity is another entity. Each node keeps a link to its parent to ease navigation. The only entities who do not have a parent are (1) the root of the AST, a special-purpose entity at the top of the tree, and (2) entities which are not part of the system's AST, because they either are not added to the tree yet, or were removed from it.

children: The collection of children of an entity. Leaves of the tree have no children. The model does not impose any restriction on the number of children.

properties: All other properties of a node are domain-specific; they are not specified in the generic model. Each node has a dictionary of key-value pairs for these properties, allowing the model to accommodate any type of property. In particular, the name of an entity is a property, independent from its identity.

Specific types of nodes are defined when adapting the model to a specific language.

3.3.2 Language Independence

We want to support several programming languages. Our model is hence generic, but specialized for the language needed. We applied our approach to Smalltalk and Java, two object-oriented languages. To support the discussion, we show an object-oriented AST in Figure 3.2, in which packages, classes, variables, methods and statements are represented.

Application to Smalltalk Smalltalk is an object-oriented, dynamically typed programming language supporting single inheritance. From the coarsest to the finest, the various types of entities we model are:

Packages: A package is the coarsest unit in a Smalltalk program. The parent of a package is the root.

Classes: Each package can contain any number of classes. In Smalltalk, each class has a superclass. The superclass is one of the properties of the class since its parent is a package.

Attributes: Each class can contain any number of attributes. Attributes are leaves of the AST: They have no children. The attributes are ordered.

Methods: Each class can have any number of methods. There is no particular order for methods in a class. Methods contain statements.

Statements: There are several kinds of statements. The most common ones are variable references (referring to a local variable, argument, instance variable or global variable),

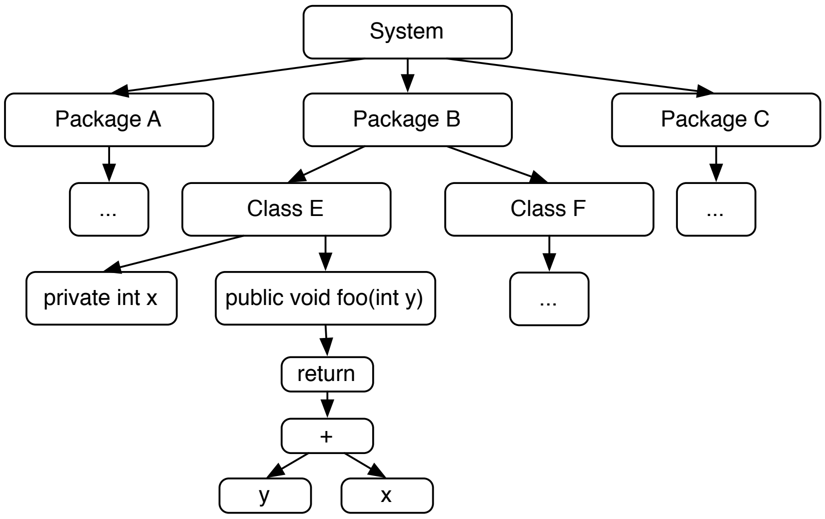


Figure 3.2: An example object-oriented program AST

variable declarations, variable assignments, message sends (*i.e.,* method calls) and return statements. Statements can either be leaves or have other statements as children. The parent of a statement is either a method or another statement.

The kind of each node (package, class, *etc.*) is a property, as well as the name. Classes have their superclasses as a property as well. Smalltalk also features method protocols, which are classifications of methods for documentation purposes. These are also defined as properties.

Application to Java The Java model is very similar to Smalltalk, with the following changes:

- Packages can be nested. A package can have both classes and packages as children. The parent of a package is either another package or the root.
- The interfaces a class implements are encoded as properties.
- The access modifiers for classes, methods, attributes, such as public, protected, private, static, final *etc.* are also encoded as properties.
- Finally, the type declarations (void, primitive types, classes and interfaces) are also encoded as properties.

Our implementation as a proof of concept does not model statements yet. This would require a full Java parser. Hence the body of a method is represented by lines.

3.3.3 Limitations

Genericity Our model is very generic; it can not easily enforce constraints on certain kinds of nodes (for example that a node can only have a limited number of children). Such constraints are implemented during the adaptation of our model to a given language.

Tree Representation Sometimes the parent/children relation is not enough to describe everything. For example, in the case of object-oriented languages we use containment for the parent/children relation. Inheritance relationships have to be encoded in an alternative way; we use properties. With languages featuring multiple inheritance, the problem would be even more prevalent.

Ordering In an object-oriented language, the classes contained in a package are not ordered, while the statements in a method certainly are. Specifying which parts are and are not ordered is one of the specialization steps.

3.4 The Change Metamodel

Our change metamodel embodies the second principle of change-based software evolution: Changes should be first-class citizens. Before diving into details and describing the changes in order of increasing granularity, we briefly list the key properties of our change model:

- Changes are transitions from one state (*i.e.,* one AST) to the next. Each change can be seen as a function taking one program state and returning a program state in which the change is applied. Our changes are thus *executable*.
- Changes also have an opposite change, whose effect when executed on an AST is to cancel out the original change. Our changes can hence be *undone*.
- At the lowest level, changes operate on a program state, that we defined as a tree: Atomic changes –as we call them– are *tree operations*.
- Nevertheless our change model supports composition in order to group low-level changes in higher-level changes. Composite changes keep the same execute and undo properties.
- Any number of AST states can coexist (created by the execution of different changes) independently of each other. Applying changes to one will not change the others.

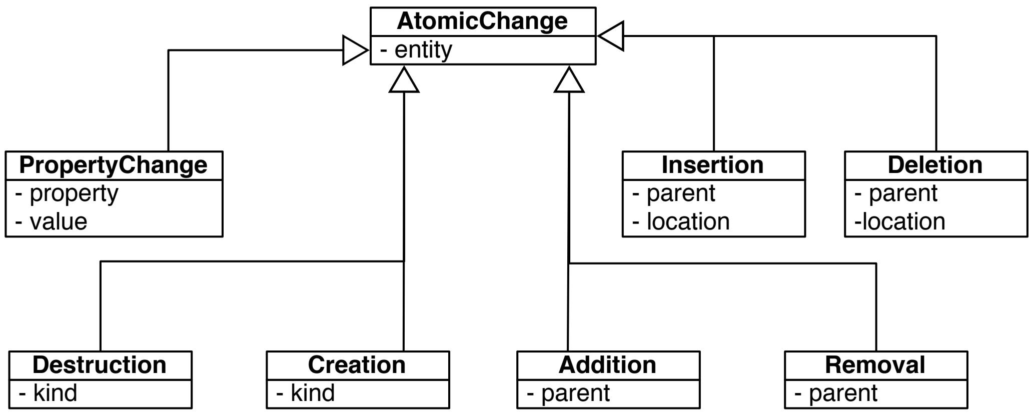


Figure 3.3: Metamodel of atomic changes

3.4.1 Atomic Changes

Atomic changes are the lowest level of changes in our model (Figure 3.3). Atomic changes are tree operations performed on the system AST. Each atomic change refers to at least the *id* of the entity it primarily affects, and keeps a link to its parent change (next section). The following tree operations are sufficient to describe any AST change (Figure 3.4):

Creation: Create and initialize a new node with id *n* of type *t*. The node is created, but is not added to the AST yet. The opposite of a creation change is a **Destruction**.

Destruction: Remove node *n* from the system. Destructions only occurs as undos of **Creations**, never otherwise (removed nodes are kept as they could be moving to a new branch).

Addition: Add a node *n* as the last child of parent node *p*. This is the addition operation for unordered parts of the tree. The opposite of an addition is a **Removal**.

Removal: Remove node *n* from parent node *p*. The opposite of the Addition change.

Insertion: Insert node *n* as a child of node *p*, at position *m* (m is the node just before n, after n is inserted). Contrary to an addition, an insertion addresses the edition of ordered parts of the tree. The opposite change is a **Deletion**.

Deletion: Delete node *n* from parent *p* at location *m*. The opposite of **Insertion**.

Change Property: Change the value of property *p* of node *n*, from *v* to *w*. The opposite operation is a property change from value *w* to value *v*. The property can be any property of the AST node, and as such depends on the properties defined in the model.

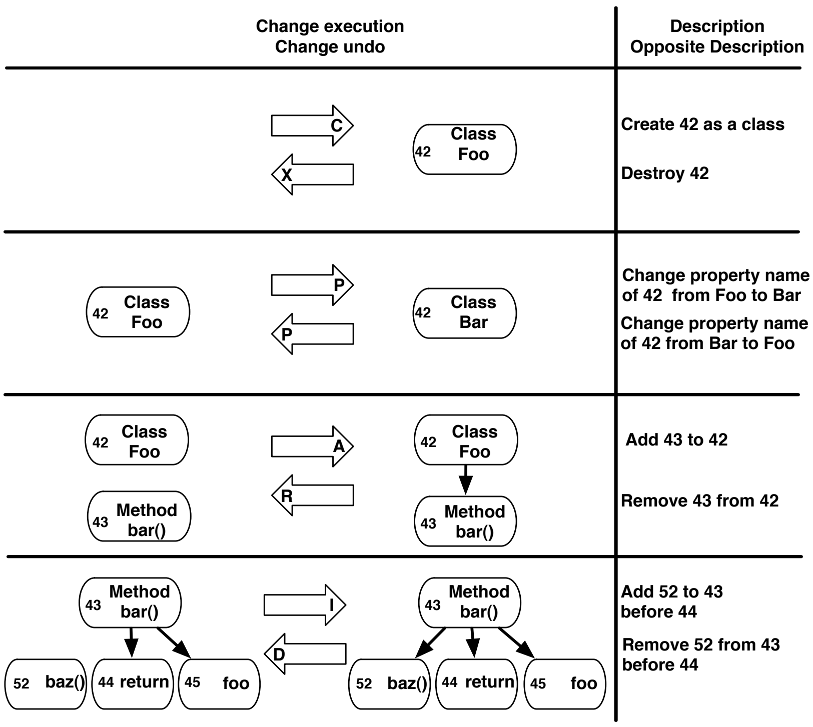


Figure 3.4: Effects of atomic changes on an AST

3.4.2 Composite Changes

Changes can be composed into higher-level changes, which keep the same execute and undo properties. Our model features several levels of composite changes.

Developer-level actions are composed of atomic changes. They represent an individual change to the system by a developer. Developer-level actions have a timestamp, and an author. Examples for the Smalltalk language are:

Create package: Contains 3 atomic changes: The creation of the package, the addition of it to the root, and the change of the package's name property.

Create class: Contains 4 or more atomic changes: class creation, addition of the class to a package, initialization of the class's name, and initialization of the class's superclass. For each instance variable added to the class, 3 atomic changes (creation, insertion, property) would also be included.

Modify class: Changes the definition of the class. It could be any subset of "Create class", excluding the change actually creating the class. It could also include the deletion of some instance variables.

Create method: Contains 3 changes for the method's creation, and any number of changes for the addition of statements in the method.

Modify method: The same as "Modify class", but for methods.

Refactorings Refactoring are behavior-preserving program transformations [Fow02]. Most are automated in IDEs nowadays. Refactorings may potentially change several places in the program, for instance if all the references to an entity are systematically updated. In our model, refactorings are composed of one or more developer-level actions. Example of refactorings in our model are:

Rename Method: Features one method modification for the method which is renamed, and one method change for each reference to the renamed method in any other method.

Extract Method: Features one method addition for the newly extracted method, and one method modification to replace the extracted code with a call to the new method in the original method.

Development Sessions A development session groups all the developer-level actions and refactorings that happened in one coding session in the IDE. We use the following heuristic to split the development history in sessions: If the difference between the timestamps of successive changes (by the same author) is more than one hour, we consider that there has been a hiatus in the development large enough to warrant starting a new session. This is the closest equivalent in our model to an SCM commit, if one assumes that developers commit at the end of each working session. We use that assumption in the following chapters.

Other possible divisions Beyond behavior-preserving transformations, our model supports more general, non-behavior preserving program transformations, as described in Chapter 7. Each transformation application results in a sequence of changes referencing the transformation they originate from.

A change may belong to several logical groups of changes. It may be part of a given development session, but also of bug fix number 12345, and of feature X. Bug fixes and features are concerns that need to have their evolution monitored as well as any program-level entities such as classes.

Thus changes can be grouped in any arbitrary way which does not match the decomposition in sessions, refactorings and development-level actions: The implementation of a feature may span several sessions, and not include every single change in it. A further example of these groupings would be a crosscutting concern.

In these cases changes can be grouped manually in a special purpose composite change, and annotated for documentation purposes. The possibilities offered by grouping arbitrary changes in this fashion are still to be explored.

3.4.3 Change histories

Our model features three kinds of change histories. Given our change description, our model of the evolution of a system is simply a list of changes. The *global change history* contains all the changes performed on the system.

For convenience, each program entity (represented by its ID) also has a *per-entity* change history, which contains all the *atomic changes* concerning each entity. From this per-entity change history, it is easy to recover the composite changes in which an entity was involved, such as all the development sessions in which it was modified.

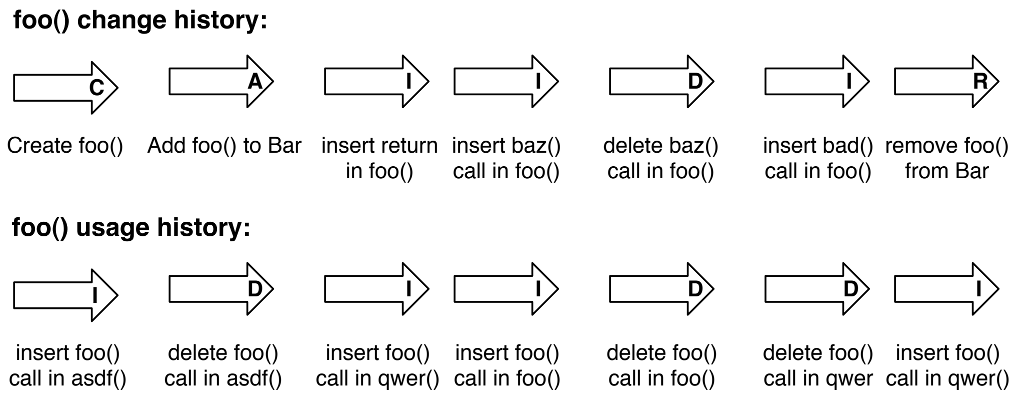

foo() change history:
C
A
I
I
D
I
R
Create foo()
Add foo() to Bar
insert return in foo()
insert baz() call in foo()
delete baz() call in foo()
insert bad() call in foo()
remove foo() from Bar
foo() usage history:
I
D
I
I
D
D
I
insert foo() call in asdf()
delete foo() call in asdf()
insert foo() call in qwer()
insert foo() call in foo()
delete foo() call in foo()
delete foo() call in qwer
insert foo() call in qwer()

Figure 3.5: Change and usage history of method `foo()`

Finally, the *usage history* of an entity refers to all the changes which increased or decreased the usage of the entity in the whole system. For example, a variable has an increased usage when a statement referencing it is inserted in the system, while a message has a decreased usage when a statement sending it is removed from the body of a method. Figure 3.5 shows the difference between entity and usage histories.

If we want to focus on a subset of the entities of the system, we can extract a *partial history* of all the changes concerning these entities. This can be easily built from the per-entity change histories of the entities in question. Some changes of other entities may be needed, such as the changes creating entities referenced by one of the entity in focus (but not all of their histories). For instance, if entity A (in focus) is added to entity B, we need several changes from B's history such as its creation and its addition to the model.

3.4.4 Generating a View of the System

Generating a view corresponds to executing part or the whole of the system's change history. This creates a system AST (or view) corresponding to the application of all the changes which were executed.

Complete view Given our change description, generating a *view* of a system at any date *d* is simple: One simply needs to execute all the developer-level actions prior to *d*. As stated in the previous section, several views of the system at different times can coexist without any problem.

Partial view Building a complete view can be quite costly if the history of the system is long. When the state of only a few entities is needed, it is possible to generate a view containing only these entities, which is much less costly. To do so, one extracts from the model the partial history needed to build the entities at a given date. Accessing the state of an entity using a partial view is much faster than if one is using a global view. Figure 3.6 shows the entities which are imported for the partial view of a method of the system. Of course, all of its statements are imported, but its parents are partially imported as well.

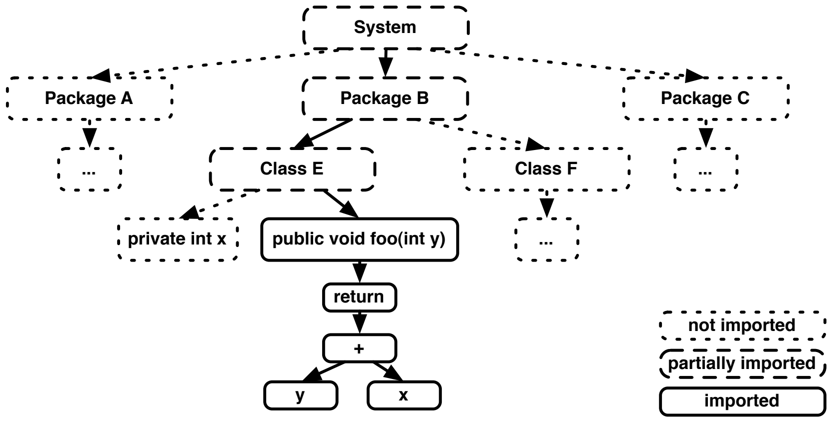


Figure 3.6: A partial view importing method *foo()*

Lazy view A lazy view is a complete view of the system whose elements are computed on request only. When an element is requested, the lazy view dynamically creates a partial view containing it, and imports it in its cache. A lazy view can also dynamically change the date in the history of the system it is viewing. This involves purging the cache, so that it can query the same entities with a new timestamp.

3.5 Recording and Storing Changes

Our third principle states that instead of being recovered from version archives, first-class changes should be *recorded* from the IDE when possible. This is the only way one can capture the actual changes performed on the system, and not merely reconstruct an approximation of them. We show the general architecture of our platform in Figure 3.7. A notifier informs our plugin of developer events. It uses the IDE's API to query relevant metadata which is added to the events. The events are either stored on disk, or directly converted to changes. These changes are then stored in a repository, from which they can be loaded and manipulated by change-aware tools extending the functionality of the IDE.

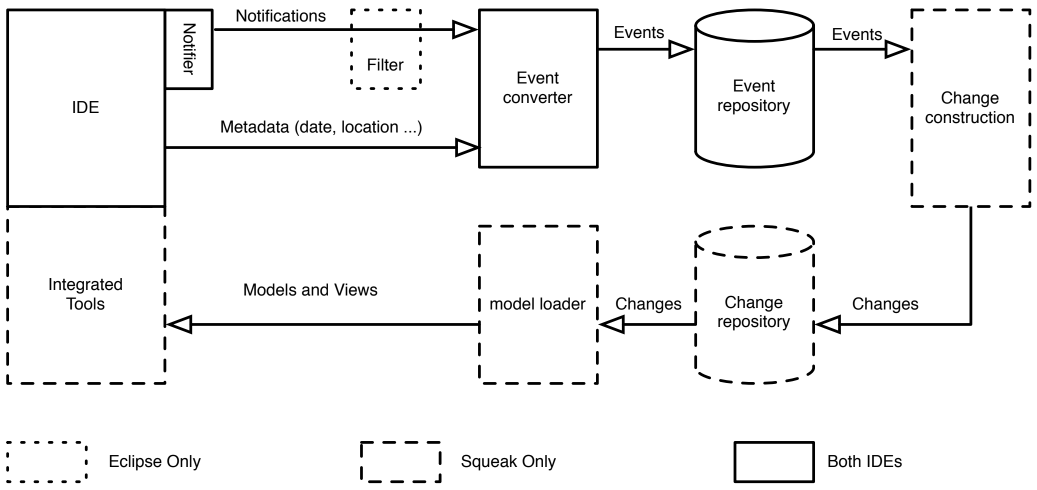


Figure 3.7: Architecture of our change-based tools

Requirements for the IDEs To monitor the changes we need to build a plugin for an IDE which is open enough for us to get the information we need. In particular, we need an IDE that provides the following:

An event notification system: The IDE should notify our plugin of events of interest, which are first and foremost where and what kind of change occurred in the system, but also when a refactoring is being performed. Knowing where, *i.e.,* on which entity a change was performed is critical as we avoid an exhaustive iteration of the entities in the system to detect which one has changed. The matching problem is greatly simplified as changing entities are known and the changes are minimal.

Access to the program representation: The IDE should answer queries about its model of the program it is editing, such as the source code of its classes and methods.

Access to various metadata: The IDE should also answer queries about who is performing the change, and at what time the change did occur.

Squeak Smalltalk Plugin The plugin we implemented in Squeak provides all of the information mentioned above, and some additional information recorded for future use: User navigation in the system, execution of code, errors and exceptions occurring during code execution, and usage of Squeak's SCM system, Monticello.

Smalltalk is peculiar since changes methods or classes must be individually *accepted*, *i.e.,* compilation of classes and methods is requested on an individual basis. The event handling mechanisms therefore issues high-level events such as "method compiled" or "class modified". This proved to be a sweet spot, as it is accurate enough, yet does not run the risk of having changes which make the system unparsable.

Eclipse Java Plugin Our current implementation of our plugin for Java is more of a feasibility study: as such, it records the information stated above, and nothing more [Sha07].

Change notification is also a bit trickier in Java. We could be notified of files being saved, but this is too coarse-grained. We chose to use keystroke notifications instead, making the Eclipse plugin notified much more often than the Smalltalk version of our plugin. We thus had to implement a filtering mechanism which groups all the successive notifications when they concern the same entity, which raised the notification frequency to the level of the Smalltalk implementation.

Model construction Constructing the model of the program's evolution based on the notifications can be performed either online or offline. When performed online, the plugin reacts to IDE notifications, queries the IDE and build the changes corresponding to the action the developer just did. It maintains a change model and an AST view at all times. When offline, the plugin queries the IDE for the necessary information and stores it in a file. That file can be read later on to build the change model.

3.6 Uses of Change-based Software Evolution

This section illustrates several usages of our model and illustrates how its various features interact. We also describe the strategies we took in order to validate our model, and outline which chapter uses which particularity of our model.

3.6.1 Example: Measuring the Evolution of Systems

We can compute two kinds of metrics: Program-level metrics, which are metrics on the AST of the system, and change-level metrics, which are metrics on the change themselves.

Program-level metrics are evolutionary metrics, which can be computed after each change in the system. Algorithm 1 shows how a metric is computed.

```
Input: Change History $M$
Output: Values of the metric
$view$ = newView(M);
$metricValues$ = Dictionary();
foreach Change ch in M do
    $view$ = execute($ch$,$view$);
    insert($metricValues$, date($ch$), computeMetric($view$));
end
```

Algorithm 1: Algorithm to compute a metric's evolution

The evolution of the metric varies with the level of change considered. One can compute it for every developer-level action, or for every development session. Higher-level groupings are also possible, such as grouping the changes by month or by year. Each metric can also be computed on subsystems (using partial histories) to get a finer view of its evolution.

Since our program representation is rich, we can compute a variety of metrics using it. Examples of program-level metrics are shown in Table 3.1, on top.

Change-level metrics are not computed on the AST of the system, but directly on the changes themselves. No AST view needs to be built and modified for each change as done above. These metrics are also applicable on any subsequence of changes, like on a set of sessions. Some of these are shown in Table 3.1, at the bottom. Furthermore, some of the program-level metrics can be computed more efficiently in this way. For instance, the number of classes can be computed not by counting the number of class nodes in the tree after each change, but with an accumulator which is incremented when the current change is a class addition, and decremented when it is a class removal.

Metric	Description
NOC	Number of classes
NOM	Number of methods
NOS	Number of statements
AMSS	Average method size (in statements)
AMSS2	Average method size (in statements, excluding accessors)
NORE	Number of references to an entity
NOAX	Number of added entities. Entities can be packages (NOAP), class (NOAC), method (NOAM) *etc.*
NOMX	Number of modified entities
NORX	Number of removed entities
ANCC	Average number of changes per children

Table 3.1: Sample program-level metrics (top) and change-level metrics (bottom)

3.6.2 Validation Strategies

The general strategy we undertook to validate our model's usefulness for both forward and reverse engineering is to define use cases in which our evolutionary information is intuitively useful, and test this hypothesis either through proof-of-concept tools or benchmarks. In all case, we use the change histories of the systems we monitored. See **Appendix A** for details.

Case studies We defined two reverse engineering approaches and one program transformation approach, all supported by tool implementations. **Chapter 4** and **Chapter 5** use visualization, and **Chapter 5** also uses metrics. **Chapter 7** presents an approach aimed at defining program transformations.

Such approaches, especially in a reverse engineering context, are hard to validate formally as they rely a lot on human judgment. Since these approaches also rely on a novel source of data, there are too many variables and not enough data points to perform a controlled experiment or a comparative study. Our evaluation was performed with case studies based on the histories of monitored programs. We plan to do comparative studies in the future when we have more data at our disposition.

When possible, we performed comparisons with SCM equivalent data, in **Chapter 4**. The other approaches rely explicitly on changes and their ordering, hence a comparison with SCM data was not possible.

Prediction Benchmarks Some problems lend themselves better to numerical validations. When the occasion showed itself, we took this strategy preferably. We found that recording a very detailed development history allowed us to easily define benchmarks in certain contexts, where we were able to assess the predictive power of our model. The general structure of such a benchmark is shown in algorithm 2.

```
Input: Change history $M$, predictor $P$
Output: Benchmark results
$view$ = newView(M);
foreach Change ch in Change history do
    $prediction$ = predict($P$);
    compareOracle($prediction$, $M$, $view$);
    process($P$,$ch$);
end
```

Algorithm 2: The benchmark's main algorithm

Of course, such a benchmark has limitations: We can only run it on the systems for which we have a recorded change history, which are not numerous, thus the result are not generalizable to every system. On the other hand, our detailed history allows us to test each system in great depth. We adopted a benchmark strategy to validate our approaches in **Chapter 6**, **Chapter 8** and **Chapter 9**.

3.6.3 What Is Used Where?

Our model was designed to support several development and maintenance tasks, so it introduces several new concepts at once. Table 3.2 shows which part of our model is used in which chapter of this dissertation. The first part displays the granularity of changes considered in each validation technique. The second part tells if they use our model extension for generic changes. The third part tells if they consider usage histories of entities. The fourth part of the table tells if the techniques used views of the state of the model, or only the change information itself. If they use views, it tells which kind of views they use. Finally, we recall the type of validation we undertook for each chapter.

	Chapter 4 Change matrix	**Chapter 5** Sessions	**Chapter 6** Coupling	**Chapter 7** Transformations	**Chapter 8** Completion	**Chapter 9** Prediction
Atomic changes		✓		✓	✓	✓
Developer actions	✓	✓	✓	✓	✓	✓
Refactorings		✓		✓		✓
Sessions		✓	✓			
Generic changes				✓		
References			✓			✓
System view				✓	✓	✓
Partial view				✓		
Lazy view		✓				
Validation	cases	cases	benchmark	cases	benchmark	benchmark

Table 3.2: Uses of various parts of the model across chapters of this document

3.7 Summary

In this chapter we described our model of change-based software evolution, aimed at supporting a wide array of maintenance tasks. We explained the principles behind the choices we took, described in details the features and concepts behind our model, and illustrated its usage on selected examples. We also outlined the validation steps we undertook.

In the next two parts, we will evaluate how comprehensive our approach really is by applying it to several problems across the reverse and forward engineering spectrum. Part II describes how our approach supports the reverse engineering of systems, *i.e.,* understanding their evolution, while Part III shows how our approach can be used to support the evolution of systems.

Part II

How First-class Changes Support System Understanding

Executive Summary

This part of the dissertation demonstrates how one can use the information gathered by Change-based Software Evolution (CBSE) in a reverse engineering context. We show that:

CBSE is useful at all levels of analysis. *In* **Chapter 4** *we showed how CBSE assists the reverse engineering of a system through visual change pattern detection and evolution scenario reconstruction. In* **Chapter 5** *we introduce a top-down process for development session comprehension. It starts with the entire history and ends with program comprehension at the individual change level.*

CBSE measures evolutionary characteristics with more accuracy. *In* **Chapter 5** *we define change-based metrics to characterize sessions based on their individual changes. Our approach is the only one that can measure these metrics. In* **Chapter 6** *we measure logical coupling with a shorter history than needed by other approaches.*

The fundamental reason behind these results is that CBSE frees us from the usual evolutionary trade-offs. Our approach tracks fine-grained entities (up to individual statements) and their individual changes. When classic views of software evolution wish to be fine-grained, they usually need to limit the number of versions they analyze. The alternative is to analyze all versions from a high level. By recording fine-grained changes instead of recovering them, we sidestep this problem.

Chapter 4

Assessing System Evolution

When dealing with an unknown system, one first needs to acquire a high-level understanding of it. Typical questions asked during that process are:

- *What are the most complex entities?*
- *What are the most changing entities?*
- *How did the system evolve to its current state?*

We present and evaluate an evolutionary visualization, the Change Matrix, which uses our fine-grained change representation to answer these questions. The Change Matrix displays evolving entities by giving precedence to the changes happening to them, rather than the successive versions of the system. The user can easily spot how the system changes and reconstruct a scenario of how the system evolved. Interactive system exploration is available: Any set of entities or time period can be explored further.

4.1 Introduction

During the first contact with a system, or when attempting to understand a rapidly evolving system, the first questions that arise are reverse engineering questions: One first uncovers high-level relationships between entities in the large amount of data available. Based on the answers to these questions, a more detailed exploration of parts of the system relevant to the task at hand is possible, *i.e.,* reverse engineering of a smaller subsystem, or actual program comprehension if the set of entities of interest has been restricted enough.

Some of these questions are best answered by analyzing the evolution of the system, rather than only its actual state. The following categories of questions are examples:

- **Complexity.** Which entities are complex? What are the important entities in the system, whose comprehension is crucial to understand the system as a whole?
- **Activity.** Which parts of the system have changed recently? Conversely, which parts of the system are stable or dead code? Are some parts of the system constantly active?
- **Crosscutting concerns.** Which changes are implemented as crosscutting concerns over several entities? Are these entities often changing together? Can one link a given functionality to one or more entities?
- **Overall evolution.** Can one outline periods in the project's evolution? Based on functionalities and periods, can one reconstruct a high-level evolution narrative of the system?

In this chapter, we investigate how much the fine-grained evolutionary history provided by Change-based Software Evolution helps in answering these questions. To that aim, we summarized our change data in a comprehensive visualization, called the Change Matrix. The change matrix displays change data according to its location, timestamp, and type. We used the change matrix on several case studies to determine how well it supports answering evolutionary questions. Further, we investigated the effect of data degradation on the answers to these questions, *i.e.,* if and how much the use of coarser-grained data as is conventionally used makes answering these questions more difficult.

Contributions. In this chapter we make the following contributions:

- The Change Matrix, a comprehensive, high-level and interactive visualization of fine-grained change data.
- A catalogue of visual change patterns based on fine-grained changes supporting the answer to the questions above.
- An evaluation of the approach on a case study.
- An estimation of the impact of fine-grained data on the quality of the answers to these questions.

Structure of the chapter. Section 4.2 presents the principles of the change matrix visualization and explains how it can answer the questions raised in the introduction. Section 4.3 presents the results we obtained from applying the visualization to one case study. Section 4.4 evaluates the impact of fine-grained data on the results, while Section 4.5 discusses our visualization and compares it with related work and Section 4.6 concludes the chapter.

4.2 Assessing Systems with The Change Matrix

4.2.1 Principles

The change matrix is a simple visualization of change information which emphasizes their type, their location in the system and their date. Additional information about the size of the changes is available interactively. The change matrix can be used to assess the evolution of a system in a given period.

Figure 4.1 shows an example of a change matrix, focused on classes and methods (a coarser-grained version focused on packages and classes is also available). The change matrix focuses on a period of the system's evolution, which is split into intervals. Intervals can be either of the same size (to emphasize periods of time), or time-warped to adapt to higher change density. The entities displayed in the visualization are classes, methods, and changes.

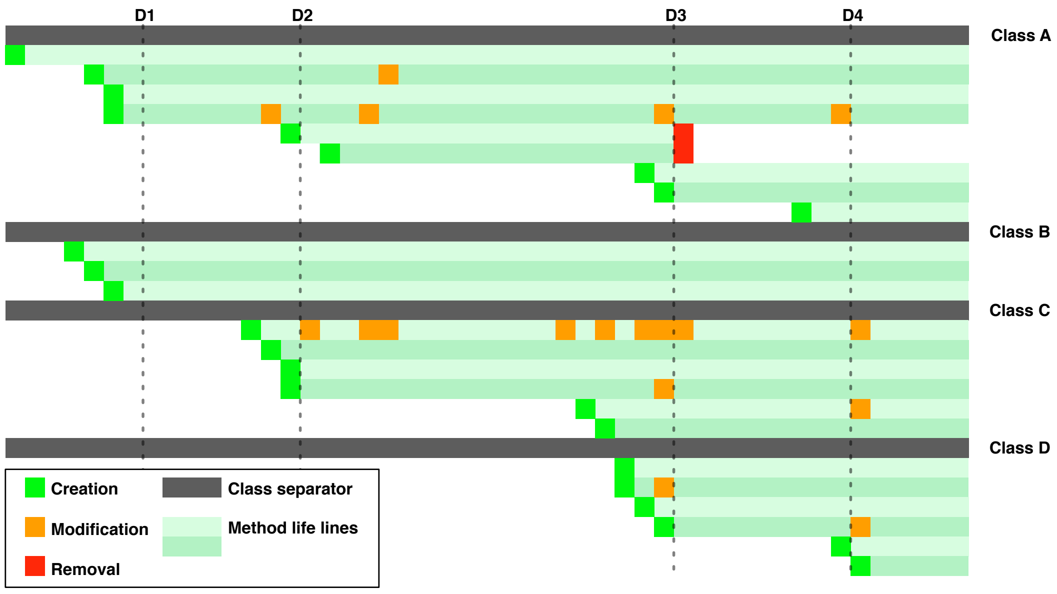


Figure 4.1: An example Change Matrix

Classes are laid out in their order of appearance, bordered by *class separators* featuring the name of the class.

Methods in each class are also laid out in chronological order, using a *life-line* figure starting at their creation and ending either when they are removed or at the end of the observed period.

Changes are displayed on the life-lines of entities. Each change is displayed as a block over the time interval during which it happened. The three main change kinds at the method level are displayed: Additions, Modifications and Removals.

The class separators can encode additional information. They can display the intensity of the changes during the interval, or the time of day they represent as a gradient of yellow and black (provided the resolution of the interval is fine enough).

When clicking on an individual change, an extra figure is created for each change in the method's history. The figure displays a finer-grained level of detail, showing the evolution of the size of the method before and after each change (Figure 4.2). The initial size of the method (in number of AST nodes) is shown on the left of each figure, and its final size on the right. If some statements are replaced by newer ones, the slope of the figure first decreases before increasing again. Clicking again reveals the actual state of the method and shows its source code before and after the change.

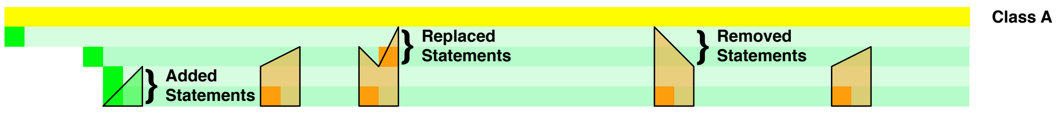


Figure 4.2: Size evolution of a method

4.2.2 Patterns

We identified several patterns which help us reply to the reverse engineering questions we formulated in the introduction. We show the patterns and explain how they can indicate the characteristics we are looking for. When possible, examples are illustrated on Figure 4.1, mentioning the classes and the date concerned. The dates of interest, D1 to D4, are highlighted with dashed lines. When method numbers are mentioned, the numbering starts at the top of the class.

Locating activity in the system. Locating activity is simple, as activity is directly denoted by the presence or absence of changes. During the initial phases of the development of the system pictured in Figure 4.1 (at date D1), classes `A` and `B` were *active*, whereas towards the end (date D4), classes `A`, `C` and `D` were active, while `B` was *inactive*.

Locating complex classes and methods. Considering the activity at a class level allows one to quickly characterize classes and methods in the system. Several patterns arise:

- *Data class.* A data class is usually small. Its method are created and are almost never –or never– changed afterwards. This is the case of class `B`.
- *Stable or dead class.* A class which has no or few recent activity. Further inspection is needed to see which of the case it is.
- *God class.* A class with a large number of methods and which has a sustained activity. Whenever the system needs to change, this class will be probably modified [Rie96]. It is critical to understand such a class to break it down into smaller, more manageable components. Class `A` fits the activity requirements, but is too small to be a *God Class*.
- *Brain method.* The equivalent of a *God Class* at the method level. It is a method which has been modified continuously. Further examination by analyzing the evolution of the size of the method is necessary to confirm the diagnostic (*i.e.,* if the method is large). A candidate would be the first method of class `C`.

Locating crosscutting changes. *Crosscutting* changes are changes that span several entities in the system. Such changes manifest themselves as vertical lines in the visualization, *i.e.,* they affect several entities in a limited period of time. They indicate functionality which is not properly compartmentalized and may be a maintenance problem [EZS+08]. An example is found at date D4, when three classes (`A`, `C` and `D`) are modified during two time intervals.

A variant is the *moving functionality* pattern, in which some entities are deleted, while others are created in another spot of the system. This denotes some refactoring efforts. Some functionality may have moved around date D3.

Finally, another pattern is *co-changing entities* denoting entities that tend to change together [GJKT97]. This happens when several entities change closely together repeatedly. The fourth method of class `A` and the first method of class `C` follow this pattern.

Locating periods in the system. We can visually identify development sessions as clumps of activity separated by periods of inactivity. We can easily see four sessions in Figure 4.1, one around each of the highlighted dates. For each session, it is easy to see at a glance which classes were concerned and to which extent.

- Session D1 seems to be a definition session, where a few methods are created but none are further modified. Further examination of their size and complexity may confirm the hypothesis.
- Session D2 is longer and contains many more feature additions.
- Session D3 is also long and features some cleanup towards the end. The first method of class `C` is constantly changed and seems central.
- Finally, session D4 seems to revolve entirely around a *crosscutting change*.

4.3 Evolution of Project I

We applied the Change Matrix to several of the histories we gathered. In the following, we only have space for one detailed report on a project's evolution. We chose project I for a detailed study, because it had the most classes in it, and was the second largest in statements. Project I is a role-playing game in which a player has to choose a character, explore a dungeon and fight the creatures he finds in it. In the process, he can find items to improve his capabilities, as well as gaining experience.

4.3.1 High-level Facts

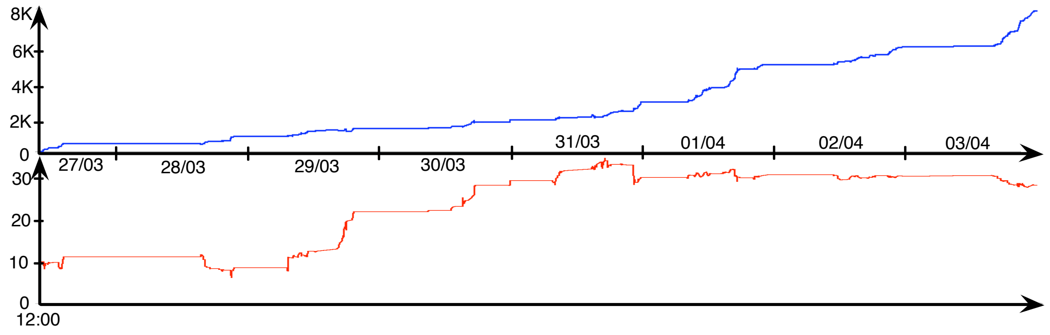


Figure 4.3: System size (top) and average method complexity (bottom) of project I

Figure 4.3 shows the evolution of two system-level metrics throughout the lifetime of the project. The unit of measure we used to evaluate project size is the number of AST nodes in the system. The projects grows regularly, with two activity spikes on the first and the third of April. On the other hand, the average complexity of the methods stays rather constant at around 30 AST nodes per method, after the evening of the 31st. This trend stays the same even towards the end of the project, where the system grows by 20 to 25% in the last hours before the deadline, reaching 8 thousand AST nodes. The slope of the system size curve is very high, only slowing down for the last 30 minutes. The constant complexity seems to indicate the project was in control until its end. Some other projects exhibited a continuously increasing complexity rate with no stabilization period.

Figure 4.4 and Figure 4.5 are the two parts of project I's change matrix. In it, intervals last 15 minutes. The first activity we perform with the matrix is to visually delimit major periods of activity in the system. To ease comprehension, these sessions are delimited by rectangles with dashed borders in both parts of the matrix. Figure 4.6 illustrates the zooming capabilities of the visualization: It displays the Change Matrix of project I focused on the class `Combat`. Since its lifespan is shorter, we can increase the resolution to five minutes per interval.

Considering the classes and their order of creation, we can see that the first parcels of functionality were, in order: The characters; the weapons; the enemies; the combat algorithm; the healing potions and finally the dungeon itself, defined in terms of rooms.

4.3.2 Reconstructing Project I's Evolution

After seeing these high-level facts about the evolution of the system, we can examine it session by session. Each session has been identified visually and numbered as shown in Figure 4.4 and Figure 4.5.

To help infer the roles of entities in the evolution, several patterns can be detected: `Hero`, `RPG` and `Combat` are *god classes*. `Items`, `Race`, `Attack`, `Minor`, `Medium` and `Greater` are *data classes*. `Mage` and `Warrior`, two character classes, experience *co-change*. *Co-change* also characterizes `Ranged` and `Melee`, two weapon classes, and `Lightning` and `Ice`, two spell classes. In this project, co-change seems to happen mainly on sibling classes, which is not as alarming as coupling between unrelated classes. Sessions 6,7,8 and 10 seem to be particularly *crosscutting*. We now explain the evolution of the project session by session.

Session 1

Date: March 27, afternoon

Goal: Data definitions

Key classes: `Hero, Spell`

The project starts by laying out the foundations of the main class of the game, `Hero`. As we see on the change matrix, it evolves continually throughout the life of the project, reflecting its central role. At the same time, a very simple test class is created (`HeroTest`), and the class `Spells` is defined.

Session 2

Date: March 28, evening

Goal: Data definitions: Professions and Weapons

Key classes: `Mage, Warrior, Weapons`

This session sees the definition of the core of the character functionality: Classes `Hero` and Spells are changed, and classes `Items`, `Mage`, `Race` and `Warrior` are introduced, in this order. Since `Spells` are defined, the students define the `Mage` class, and after that the `Warrior` class as another subclass of `Hero`. This gives the player a choice of profession. The definitions are still very shallow at this stage, and the design is unstable: `Items` and `Race` will never be changed again after this session.

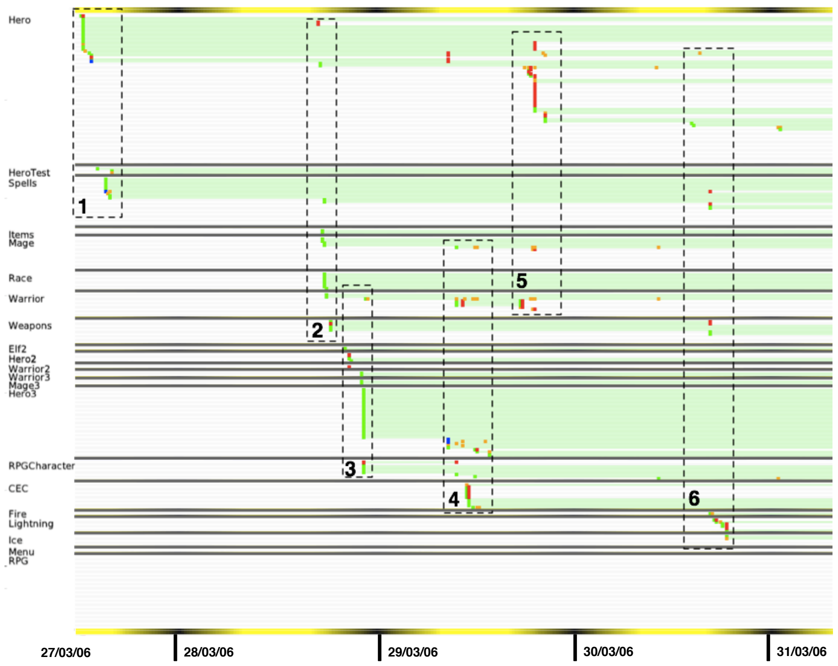


Figure 4.4: Change matrix of project I, 27/03 to 31/03

Session 3

Date: March 28, night
Goal: Alternative character definitions
Key classes: `Hero3`

This session supports the idea that the design is unstable, as it can be resumed as a failed experiment: A hierarchy of races has been introduced, and several classes have been cloned and modified (`Mage2`, `Hero3` *etc.*). Most of these classes were removed, or kept as dead code.

Session 4

Date: March 29, afternoon
Goal: Character functionality transfer
Key classes: `Mage, Warrior, Hero3`

This session is also experimental in nature. Several classes are modified or introduced, but were never touched again: `Hero3`, `CEC` (where several methods are added just to be deleted, indicating renames), `RPGCharacter` (except two modifications later on, outside real coding sessions). `Mage` and `Warrior` are changed too, indicating that some of the knowledge gained in that experiment starts to go back to the main branch.

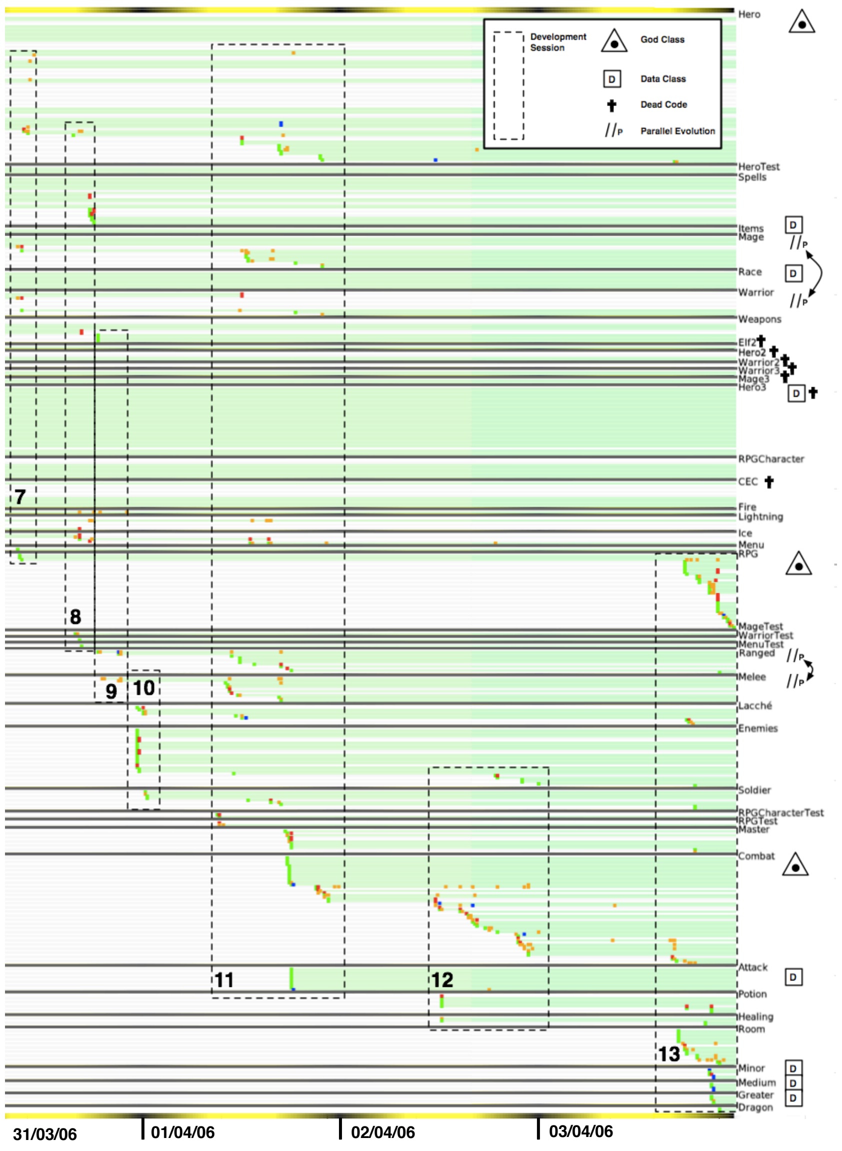


Figure 4.5: Change matrix of project I, 31/03 to 03/04

Session 5
Date: March 29, evening and night
Goal: Character functionality transfer
Key classes: `Hero, Warrior, Mage`
This session achieves the knowledge transfer started in session 4. `Hero` is heavily modified in a short period of time, including massive renames. In the following sessions, `Hero` will regain some stability. In the meanwhile, `Mage` and `Warrior` are consolidated. Already with sessions 2, 4, and 5, we can see that `Mage` and `Warrior` are *co-changing* classes.

Session 6
Date: March 30, late afternoon
Goal: Weapon and spell diversification
Key classes: `Weapons, Spells, Lightning, Fire, Ice`
This session sees a resurgence of interest for the offensive capabilities of the characters. A real Spell hierarchy is defined (`Lightning, Fire, Ice` are subclasses of `Spells`), while the `Weapons` class is modified as well. Prior to that, `Hero` is slightly modified, confirming its *god class* status, as each change to the system seems to involve it.

Session 7
Date: March 31, noon
Goal: Game class definition
Key classes: `RPG, Hero, Mage, Warrior`
The first full prototype of the game. The main class, `RPG` (standing for Role Playing Game) is defined, as well as a utility class called `Menu`, proposing menu-based choices to the player. `Mage`, `Warrior` and their superclass `Hero` are modified as well, strengthening the patterns we already established.

Session 8
Date: March 31, evening
Goal: Testing and spells
Key classes: `Spells, Lightning, Ice`
This session features some work on spells, considerably changing the root class `Spells`, and its subclasses `Lightning`, `Fire` and `Ice`. In parallel, several simple test classes, `MageTest`, `WarriorTest` and `MenuTest`, are created.

Session 9
Date: March 31, night
Goal: Weapon diversification
Key classes: `Weapons, Ranged, Melee`
There was no real separation between this session and the previous one time-wise. However the entities in focus clearly change rapidly, hence we separated session 8 and 9 for clarity.

This is an example of a fluid transition from one functionality to the next which might not be reflected in the SCM system data if the code is not committed before the transition.

This session focuses on weapon diversification with classes `Melee` and `Ranged`, both subclasses of `Weapons`; these classes have a very close evolution (*co-change*) for the rest of their life, as their patterns are really similar, in the same way `Lightning` and `Ice` co-evolve constantly .

Session 10

Date: March 31, night
Goal: Enemy data definition
Key classes: `Enemies`

This session also features a fluid transition with the previous one. The student's rhythm of work is intensifying, and several features are being worked on at the same time.

A real hierarchy of hostile creatures appears: `Enemies`, `Lacché`, and `Soldier`. The system is a bit unstable at that time, since `Enemies` has a lot of methods added then removed immediately, suggesting renames.

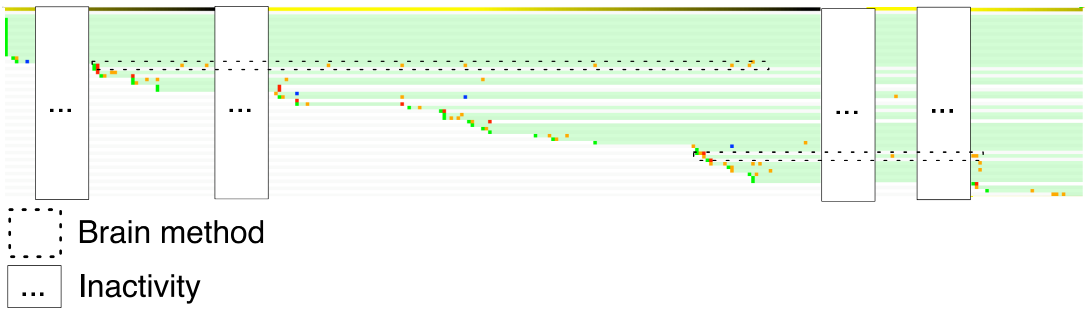


Figure 4.6: Change matrix zoomed on the class Combat

Session 11

Date: April 1st, noon to night
Goal: Combat algorithm definition
Key classes: `Weapons, Combat, Hero`

As the deadline for the student project approaches, sessions become longer. Work intensifies further, and transitions between activities are not as smooth as previously. We can however distinguish two phases in this session: Before, and after the definition of class `Combat`.

This intensive session sees the first iteration of the combat engine. The weapons, spells and characters (heroes as well as the `Enemies` hierarchy) are first refined. In each case, the co-change relationships are strengthened, as the parallel evolution of `Lightning` and `Ice`, on the one hand, and `Ranged` and `Melee`, on the other hand, is easy to see in the session. `Mage`, `Warrior` and `Hero` are also subject to co-change. Then a new enemy, `Master`, is defined.

The implementation of the `Combat` class shows a lot of modifications of the `Weapons`, `Spells` and `Hero` classes, indicating some *crosscutting*. An `Attack` class soon appears. Judging from its (non-)evolution, it seems to be a *data class* with no logic, comforting the idea that `Combat` is a *god class* using it. After theses definitions, the implementation of the real algorithm begins. We see on Figure 4.6 –the detailed view of `Combat`– that one method is heavily modified continuing in the next session. It seems to be the heart of the combat algorithm.

Session 12

Date: April 2nd, noon to night
Goal: Combat algorithm continued
Key classes: `Combat`

Development is still heavily focused on the `Combat` algorithm. Compared to the previous session, we observe that the modifications are much more localized. This session also modifies the main combat algorithm, and at the same time, two methods in the `Hero` class, showing some coupling between the two *god classes*. In parallel, `Enemies` is also changed, furthering the integration of both kinds of characters in the combat.

It is interesting to note that the subclasses to `Enemies` and `Hero` do *not* change in this session. This indicates that either the hierarchies are not fragile, or that `Combat` handles everything, using the other classes as *data classes*. Considering the evolution of `Combat`, we are inclined to think the latter, but only a closer code inspection involving actual program comprehension could tell.

A second method featuring a lot of logic is implemented, as shown in Figure 4.6: several methods are often modified. This method, along with the one introduced in the previous session, seem to be the *brain methods* handling the combat algorithm. In parallel, classes `Potion` and `Healing` are also defined, allowing the heroes to play the game for a longer time.

Session 13

Date: April 3rd, afternoon to night
Goal: Main game loop and dungeon definition
Key classes: `RPG, Combat, Room`

This last session has a wider focus than the previous one, as it ties “loose ends” in a time-limited project.

The students finish the implementation of `Combat`, changing the `Enemies` hierarchy in the process. This change seems like a change to a polymorphic method since the change is spread out on each hierarchy class but performed quickly. A significant amount of methods are changed in the `Combat` class, but only in the methods defined last. Either functionality has moved to these methods, and the older methods are no longer used, or the algorithm is compartmentalized, the latter methods being concerned with enemies rather than characters of the `Hero` hierarchy. A finer code inspection is needed to determine which hypothesis is accurate.

After finishing `Combat`, this session also resumes the work on the entry point of the game, the `RPG` class. Only now is a `Room` class introduced, providing locations in the game, an aspect overlooked until then. These classes are tied to `Combat` to conclude the main game logic. To finish, several types of potions –simple *data classes*– are defined, and a final monster, a `Dragon`, is added at the very last minute.

4.3.3 Recapitulation

From a high-level analysis of the evolutionary data we recorded of small-scale project, we were able to observe the following:

- The average complexity of the system increased at first, but was kept in control after a third of the project's evolution, even if the system size increased significantly.
- We deduced the role of classes from a cursory observation of the patterns we detected in their evolution. In particular, we identified which classes were *god classes* (`Hero, Combat, RPG`), *data classes* (`Potion` class hierarchy, `Attack`), *dead code* (`Hero2, Mage2, Hero3, CEC`), and the reason of its presence (experimental character definitions), and which classes had *co-change relationships* (`Ranged` with `Melee`, `Lightning` with `Ice`, `Mage` with `Warrior`, classes in the `Enemies` Hierarchy).
- We reconstructed how the project evolved based on its change history and the patterns we discovered. We described when and where in the system each functionality was defined, and identified the most fragile functionality (combat between parties, which spans several large classes), and have a clear picture of the high-level relationships between functionalities.
- We formulated a handful of hypotheses that could serve as program comprehension starting points, concerning the hierarchy of `Hero` and `Enemy`, and their relation to the `Combat` class.

In short, we gathered a reasonable idea of the system's design, functionality and evolution based on the analysis of its change history. We are aware of its shortcoming and of the probable locations one need to change in order to alter a high-level functionality. We thus think our initial reverse engineering effort of this system was successful.

4.4 Impact of Data Degradation

We have shown on one example how a review of fine-grained changes allows us to infer high-level facts about the design and evolution of a software system. But how much is this due to the fine-grained data our approach provides?

In order to evaluate how much our approach depends on fine-grained data, we simulated the application of the same visualization on data degraded to match the granularity of data found in SCM systems. We then evaluated how well the patterns we previously located were preserved when analyzing degraded data. We proceeded in two steps: We first simulated the usage of SCM commits instead of change recording, and then evaluated the effect of data sampling, *i.e.,* intentionally reducing the number of versions analyzed in order to deepen the analysis on each one.

We simulated SCM commits by using the assumption that each development session would have ended with a commit to the versioning system. Since the Change Matrix displays changes according to their time, location and type, displaying commits in the change matrix amounts to altering the time stamp of each change belonging to the same session so that they share the same time stamp, considered to be the commit time (in our case, the last change's time stamp). Figure 4.7 shows a side-by-side comparison of a subset of the Change Matrix of Project I, with and without data degradation. We see clearly that some of the patterns, in particular those involving repeated modifications to the same entity, are much harder to locate. In particular, co-changing methods in the same session are no longer visible, as changes to other methods seem to appear at the same time. Also, crosscutting changes (for examples, the last changes on `Ranged` and `Melee`, or some of the last changes in `Combat`) are no longer distinguishable from sequences of individual changes, and repeatedly changed brain methods are no different from methods changed once per session.

This comparison is still advantageous to commit-based evolution analyses, since a precise modeling of systems (such as ours, which is at the AST level) is usually performed on a subset of the available versions. We simulated version sampling by grouping sessions in sets of four, and altering the dates of the changes in each session so that all changes share the same time stamp. This approximates a sampling in which 25% of the versions are kept. For reference, Raţiu *et al.* [RDGM04] kept around 10% of the commits, while Xing and Stroulia [XS05] kept one version per month. In that case, patterns are even harder to locate: Only the most obvious activity patterns (mainly at the class level), remain detectable, as the mass of changes happening at the same time hides all the other patterns. Figure 4.7 shows the Change Matrix (on top) and the sampled Change Matrix (at the bottom). The co-change-relationship between `Melee` and `Ranged` which is obvious on top looks now no different than their relationship with the unrelated `Lacché` class. If anything, `Lacché` and `Ranged` share a single change in the last session, making them slightly more related than `Melee` and `Ranged`. Finally, the observations one can make on the `Combat` class are now very limited: One can only infer that it is a large, fast-evolving class.

4.5 Discussion

Comparison with Related Work

Most evolutionary visualizations display each versions individually. Few visualizations display several versions of a system simultaneously, in order to review the overall evolution of a system or part of it. All these approaches share the limitation that the data they considered was extracted from text-level, version-based SCM. They are limited to changes to files across versions, while the data used by the Change Matrix is finer-grained. As such, all of these approaches suffer from the problems we outlined in the previous section.

Lanza's Evolution Matrix [Lan01] displays the evolution of classes by laying out their successive versions in a row. Classes are ordered by order of apparition. In essence, the Change Matrix is a finer-grained version of the Evolution Matrix as it can display the evolution of methods and can display changes between versions.

Revision Towers by Taylor and Munro [TM02] showed several versions of the same file in the same figure, as levels of a tower which allowed to see the evolution of a given file across its lifetime at a glance.

Wu's *et al.* Evolution Spectrographs [WHH04] can be set up to display –as the Change Matrix does– the changes rather than the successive versions of the entities they consider. However Evolution Spectrographs has been only applied to versioning systems so far.

A finer-grained evolutionary visualization is a polymetric view defined by Gîrba *et al.*, the Hierarchy Evolution Complexity View. It displays the structure of the system in terms of class hierarchies, but overlays the evolutionary information such as the age of classes or inheritance relationships on top of the structure [GLD05]. The changes themselves are not displayed: Only the latest version of the hierarchy is, with deleted classes and inheritance relationships displayed in cyan.

Generalizability

Since we report only on case studies, we can not generalize our results to every system. One might argue that the results depend on the style of the developer. We kept our description of the patterns intentionally generic, so they can apply in other contexts. The patterns appeared in the other case studies we applied our approach to (the other student projects, SpyWare), so we believe they would appear on any system.

Scaling Up and Down

The example we reported on was one of the largest student project we examined (totaling 41 classes), but is still small by any standard. The visualization we presented has however the potential to scale. If higher-level insights are needed because of the large number of entities, the same visualization can be applied to packages and classes, reducing the space the visualization takes on the Y axis. If the time period is long, the intervals used to display changes can be made longer, reducing the space taken on the X axis.

Once system-level questions are answered, the same visualization can be used on shorter periods and reduced number of entities. If the Change Matrix is used on a given period, it will automatically omit the entities which do not change during the period. If the Change Matrix is used on a given entity, the period in which it does not change can be omitted or condensed with time warping. These factors allow the Change Matrix to be brought to the level we used it even for larger projects. Other approaches might not support increasing the level of detail up to a point: Ours ensures we can focus on the smallest level of detail if needed.

Unincorporated Data

The Change Matrix does not use all the data our model records. Two additional data sources could be incorporated.

Refactorings are changes which do not alter the behavior of the system, yet span several entities. These could be mistaken for other crosscutting changes. In our case study, no refactorings were performed, so this did not apply. For other change histories we might want to distinguish refactorings from other changes.

The usage of entities is not incorporated. One could imagine displaying it on the life-line of the entity, making it darker as the entity is used more widely. Since this might clutter the visualization, this could be an interactive feature. Having this information should help distinguishing between stable (increasing usage) and dead code (decreasing usage), and identify the central points of the system (a very complex class might actually be on the fringe of the system). Displaying which entities are using the entity in focus would help determining if an entity is public (*i.e.,* used system-wide) or private (*i.e.,* only used in its package).

4.6 Summary

During the initial assessment of a system for reverse engineering, visualizing the evolution of the system allows one to characterize parts of the system according to how it changed in the past. To assist this task, we defined a comprehensive evolutionary visualization using our change data, the Change Matrix. The Change Matrix focuses on the changes performed in the system rather than successive versions as they would be extracted from a versioning system. Visualizing such fine-grained changes allowed us to easily characterize parts of the system according to several dimensions:

- **Complexity**. Classes can be labeled as stable, data or dead classes when they change rarely, or as god or complex classes if they change often.
- **Activity**. It is trivial to identify which parts of the system have been active at any point in time.
- **Crosscutting concerns**. Crosscutting or moved functionality is visually easy to assess, as well as co-changing entities.

- **Evolution**. The order of appearance of functionality is easily accessible, and the system's evolution can be reconstructed.

We identified several patterns based on the occurrence or absence of fine-grained changes at a given point in time. Localizing these patterns on a change-centric visualization allows one to reconstruct with a modest effort an evolution narrative of the system, based on the change matrix and the names of classes and methods changed.

We showed that the fine-grained change data produced by Change-based Software Evolution answers high-level reverse engineering questions about software systems by characterizing the evolution of systems with patterns and providing access to a detailed change history. Even for high-level questions, the quality of the data is primordial: We compared our results with what can be obtained by using data equivalent to the one found in an SCM system and showed that the results were much less accurate. This problem is further compounded by the common practice of sampling the data in order to make the analysis of individual versions more precise, which increases the amount of changes between each versions. In that case, most of the patterns we identified are no longer detectable.

In short, the data Change-based Software Evolution provides eases the reverse engineering of systems by providing access to a comprehensive and accurate history of the changes encountered by domain-level entities in the system. The accuracy of these changes improves the insights one can get on systems even on a high level, as trends and patterns of relationships between entities can be detected with a much greater accuracy when the history of these entities is closely tracked. Since the data provided by CBSE is fine-grained, the level of data considered can be scaled up or down.

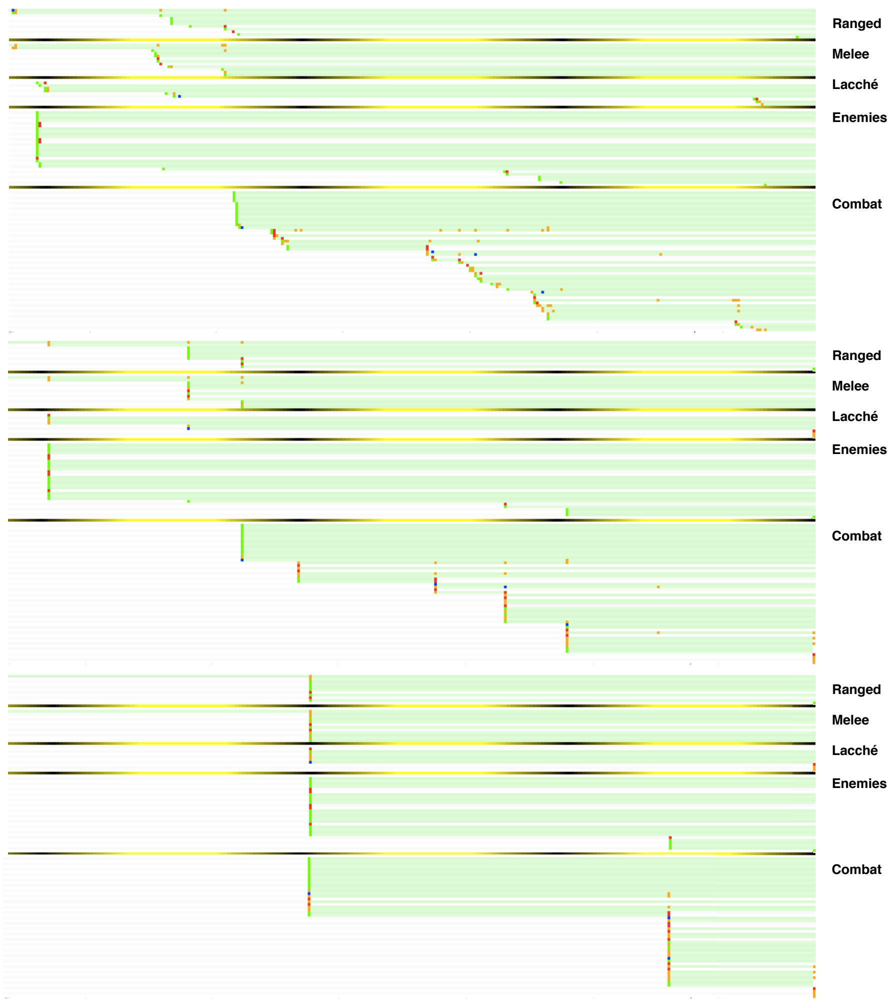


Figure 4.7: Impact of data loss: Original (Top), Commits (Middle), Version Sampling (Bottom)

Chapter 5

Characterizing and Understanding Development Sessions

What happens during a development session? Mining a versioning system's archives does not tell us the whole story as only the outcome of a session is stored. In contrast, change-based repositories contains the exact changes *that were performed during any development session. Accessing the changes in the sequence they happened helps the fine-grained comprehension of the activity carried out during a development session.*

We first provide a high-level context for session understanding by defining a characterization of sessions. This session characterization is based on metrics measuring particular aspects of the changes that occurred during the session, and distinguishes between various types of development sessions.

Further, we support program comprehension through a top-down session exploration process. This process is based on the previous characterization and uses several interactive visualizations. It allows a developer to choose and explore the sessions –and the changes contained within them– in order to better understand how a given system functionality is implemented by understanding it incrementally. *The developer is free to navigate between high-level views of the system (where the unit of change is the session), and lower level views of the system (where the unit of change is the individual change to methods and classes of the system).*

5.1 Introduction

When a change to a system is needed, the first thing to do is to localize where in the system the change should be made. We have shown in Chapter 4 how Change-based Software Evolution helps to locate functionality to support such a task.

The next task is then to understand the few entities that collaborate to achieve the functionality that one needs to change. Understanding the whole program is unnecessary, but understanding selected source fragments is critical [ES98]. However, even understanding a relatively small subset of the code of a system is difficult if one does not start at the right place or does not follow the right path through it.

A possible path to take is the one the developer himself took while implementing the functionality. Such a path might contain mistakes and indirections, but developers always follow a certain logic when writing code and proceed incrementally. Change-based Software Evolution allows access to the implementation process as it records *entire* programming sessions. Following the programmers' footsteps as an aid to program comprehension becomes possible by reviewing each change in the system in order.

This information can not be recovered *at all* from a traditional versioning system since only the outcome of the session will be committed to the SCM system. On the other hand, IDE monitoring approaches only keep a shallow model of what the developer has actually done (usually navigation information and shallow change information), which is not enough to reconstruct the actual changes the developer did. Only recording changes provides enough information.

In this chapter, we describe how program comprehension can be assisted by reviewing development sessions. Our session-based program understanding approach is based on two steps. We first enrich the context of the session by providing a high level characterization of the activity in the session, based on change metrics. This characterization provides high-level additional insights on what actually happens during the session. Second, in order to support actual program understanding, we define a session understanding process supporting top-down exploration, from several sessions at the same time down to individual changes to program entities.

Contributions. The contributions of this chapter are:

- A characterization of development sessions across several dimensions, enriching the context a programmer has for understanding a session.
- The definition of several change-based metrics, unique to our approach, which constitute the basis for our session characterization.
- The definition of a session exploration process supported by tools and interactive visualizations which allows one to review a set of sessions and the individual changes in a session in order to understand the functionality implemented during these sessions.

- A validation of these techniques on selected sessions across two case studies, featuring several hundreds development sessions.

Structure of the chapter. Section 5.2 motivates the usefulness of session-based program understanding and characterization. Section 5.3 presents our characterization and the change-based metrics we used to define them. We then describe the process –and the tools that support it– in order to select and understand relevant sessions in Section 5.4. Section 5.5 validates our approach on two case studies comprising hundreds of development sessions. We describe in detail selected development sessions, and how our approach assist their understanding. Section 5.6 discusses our approach and related work, while Section 5.7 concludes the chapter.

5.2 Motivations for Session-based Program Understanding

We have already seen in Chapter 4 the usefulness of analyzing fine-grained changes sequentially on several occasions. We were able to:

- **Highlight co-change relationships.** Entities closely related to each other are usually changed together. For program understanding, it makes sense to review changes to related entities at the same time. Some of these relationships, for example those relying on side effects, might not be obvious as there may not be a direct reference from entity to the other. The original programmer will be aware of the relationship and change both, making the relationships more obvious.

- **Differentiate functionalities.** In the previous chapter's case study, one session was graphically divided in three distinct sub-sessions (Sessions 8, 9 and 10). The graphical differentiation was obvious as three distinct areas of the system were changed in sequence, but modifications were local to each area. However, a single commit might have been performed merging these three tasks. Understanding these changes without accurate historical information might lead one to believe that they are closely related. In reality, they were not. Starting with a wrong hypothesis makes understanding harder, as the maintainer will try to relate these distinct pieces.

- **Contextualize changes.** The changes surrounding an individual change give insights about its aim. In the previous chapter we were able to visually identify method renames (methods being deleted while other methods in the same class appear simultaneously), or potential moves of functionality (changes and deletions in a class while methods are added to another class shortly after). In this chapter, we inspect these changes more closely and additionally incorporate actual refactorings which our approach records. Knowing that a change was performed automatically by a refactoring tool is helpful, since the change is guaranteed to be behavior-preserving: A closer inspection is not needed.

- **Incremental implementations.** The previous chapter's case study showed this particularly towards the end, in the implementation of the `Combat` algorithm. The main concepts of a feature are defined first. Later, secondary concepts are defined and primary concepts are refined. If a feature implementation is reviewed according to its timeline, a basic version of it can be reviewed initially. Only after the general feature is defined, improvements such as optimizations, peculiar cases and generalizations are implemented. Following the steps of the developer leads to a more natural and progressive understanding of the code.

We see that there are several reasons to understand programs sequentially as they are built. Some of these apply also to characterizing sessions:

- **Increasing context.** Characterizing sessions gives an overall context to a session or parts of it. Context allows us to better understand changes, be it because we know which entities are related to the one being changed, or because certain changes belong to refactorings. In the same fashion, an overall session characterization adds context to the session. Knowing that a session is refactoring-dominant makes it different from a bug-fixing, feature addition or feature enhancement session.

- **Characterize entities.** We have not investigated this, but we believe program entities can be characterized by the development sessions they are involved in. An entity often involved in maintenance related sessions is either a very central piece of the system (potentially a god class), or may have a significant amount of defects. Both cases invite a closer inspection of the entity.

- **Focusing the Reviewing and Testing Effort.** Code reviewing is an established practice to prevent defects, but resources might be limited. Generalizing the previous point, one can allocate more resources to code originating from sessions with a higher risk of containing bugs. Several change metrics could indicate this, such as the propensity during a session to move to seemingly unrelated entities (potential side effect), or seeing entities being changed repeatedly.

5.3 A Characterization of Development Sessions

To provide more context when understanding sessions, we characterize them according to metrics we defined. We first explain the dimensions we chose for the characterization, then present the change-based metrics we used as a basis for the characterization. We then perform a quantitative analysis of the session characteristics on our case studies.

5.3.1 Primary Session Characterization

We characterize each session according to several dimensions. The primary characteristics are the session **Architectural Type** and its **Duration**. Since the characterization is based on change-level metrics, it is applicable to any sequence of changes. As such, it is also useful to characterize smaller (phases in a development session) or larger (the set of sessions related to an entity) groups of changes. This allows a session to be separated in several phases if it helps its understanding.

Architectural Type. The primary dimension is the type of activity carried during a session. To create a concise but effective vocabulary when we talk about the different types of sessions, we use a metaphor taken from Brant's "How Buildings Learn" [Bra94], where he describes buildings as multilayered structures where inner layers change faster. Brant's book is about architecture and therefore his layers are (from inner to outer) stuff, space plan, services, skin, structure, and site. The idea is that for instance "stuff" (the furniture) is changed more often than the space plan of a house, which is also changed more often than its skin, *etc.*

We reuse that metaphor for software development since the frequency of various development activities vary in the same fashion, and the types of activity can be mapped to architectural terms. The possible types of session across the architectural dimension are:

- **Decoration** is the smallest and most common kind of activity. In our case, it corresponds to light maintenance activity, such as corrective maintenance. It is characterized by slight alterations to the code base, such as changing method bodies. Pure decoration sessions do not add any new methods.
- **Painting** is the next most common activity. It corresponds to feature refinement, *i.e.,* extension or alteration of an existing feature. Painting is characterized by the addition and the modification of methods on already existing classes.
- **Masonry** is active construction of the system and refers to addition of new functionality in the system. Since in an object-oriented system the class is the unit of behavior, we define this as adding –or changing– both classes and methods to the systems.
- **Architecture** is groundwork for further construction and corresponds to addition of major features. An architectural session adds a large number of new classes and may adds packages to the system.
- **Restoration** refers to preventive maintenance of the system, and is linked with refactoring actions, such as actual refactorings or movements of functionality.

Duration. We qualify each session according to its length, in five categories: **Blitz** sessions have a very focused activity, and last less than 15 minutes; **Short** sessions last between 15 and 45 minutes; **Average** sessions last between 45 and 90 minutes; **Long** sessions last between 90 minutes and 4 hours –an entire morning or afternoon of development work; **Marathon** last more than half a day of work and regroup all sessions lasting more than 4 hours.

Conceivably, long and especially marathon sessions could be more error-prone since fatigue has time to set in. On the contrary, blitz and short sessions indicate a focused activity that was planed and delimited in advance.

5.3.2 Session Metrics

We first describe the change-based metrics we defined, before explaining how we use them to characterize sessions. Our metrics are explained in table Table 5.1. Metrics in bold are only obtainable through change recording, and not through recovering changes from SCM archives. Not all metrics are used to compute the primary session characteristics. Others are used by themselves when the session is inspected, as *secondary session characteristics*.

Metric	Metric Description	Indicator of
SLM	Session Length – expressed in minutes.	Primary **duration** characteristic.
TNC	Total Number of Changes, *i.e.,* developer-level actions performed during a session.	Amount of work actually performed in a session.
TSC	Total Size of Changes, *i.e.,* number of atomic changes performed during a session.	Amount of work actually performed in a session.
NR	Number of (recorded) Refactorings, and related actions.	Preventive maintainance
SA	Session Activity, *i.e.,* changes per minute ($SA = \frac{TNC}{SLM}$).	Speed at which the task was performed.
NAM	Number of methods added during a session.	Amount of new behavior
NCM	Number of methods changed during a session.	Behavior refinement
UNCM	Unique number of methods changed ($UNCM \leq NCM$). Same as NCM, except every method is only counted once.	Behavior refinement
NTM	Number of touched methods, *i.e.,* methods that were modified or added.	Change amount.
ACM	Average changes per method ($ACM = \frac{NCM}{UNCM}$).	Incrementality
MCM	Most changed method, the highest number of changes applied to a single method during the session.	Presence of outliers.
NAC	Number of classes added during a session.	Behavior extension.
NCC	Number of classes whose definition changed during a session, *i.e.,* with addition/removals of attributes	Behavior extension.
UNCC	Unique number of classes changed ($UNCC \leq NCC$). Same as NCC, but each class is counted only once.	Behavior extension.
NTC	Number of touched classes, *i.e.,* classes that were modified or added.	change magnitude.
NIC	Number of involved classes, *i.e.,* classes that were added, changed, or who had a method added, or changed.	Extent of the changes in the system, crosscutting.
ACC	Average changes per class ($ACC = \frac{TNC}{NIC}$).	Crosscutting
MCC	Most changed class, the highest number of changes applied to a class.	Presence/absence of outliers.

Table 5.1: Session Metrics.

Our primary characterization of metrics is based on detection strategies [LM05]. Detection strategies are combinations of metrics and thresholds detecting higher-level characteristics of software system. Their primary use is detecting design flaws [Mar04].

Choosing the thresholds is an important part of designing a detection strategies. Thresholds can be absolute, or relative to the project they are used on. Since we do not yet have a large enough amount of data to determine thresholds empirically, we also use metrics on their own as a secondary characterization. In that case, their value is accompanied with a percentile telling their relative standing in the project. This also accounts for the variation in styles of developers. Table 5.2 shows the combinations of metrics and thresholds used in our primary characterization.

Characteristic	**Description**	**Rationale**
Decoration	$\frac{NCM}{TNC} \geq 0.66$	Two-thirds or more of the changes are method modifications.
Painting	$\frac{NAM}{TNC} \geq 0.33$	One-third or more of the changes in the session introduce a new method.
Masonry	$NAC + NCC > 0$	At least one class is added or modified. Masonry is superceded by Architecture.
Architecture	$NAC + \frac{NCC}{2} > 5$ or $NAP > 0$	At least one package, or more than 5 classes are added. Modifying a class counts as half as much.
Restauration	$NCR + NMR > 6$	A large amount of refactorings is performed during the session.
Blitz	$SLM \leq 15$	A very short session.
Short	$SLM > 15$ and $SLM \leq 45$	A short session.
Average	$SLM > 45$ and $SLM \leq 90$	An average session, in which a normal task should be completed.
Long	$SLM > 90$ and $SLM \leq 240$	A longer than usual session.
Marathon	$SLM > 240$	A session longer than a single morning or afternoon, denoting intense work.

Table 5.2: Definition of our characterization.

5.3.3 Quantitative Analysis of the Characterization

We looked at two of our case studies which have a large number of sessions. These are SpyWare (around 500 sessions at the time the analysis was done), and Project X (around 120 sessions).

Table 5.3 presents high-level primary characteristics of the sessions for both projects. Each session is characterized by its architectural type and duration. The session types are not mutually exclusive, *i.e.,* a session can be of more than one type, such as Masonry and Painting.

The table reveals that some session types tend to have a characteristic length: Architecture and Restauration are longer sessions, while the highest proportion of Masonry sessions is found at average lengths. Painting and Decoration sessions are rather homogeneous in SpyWare, but occur more in shorter sessions in Project X. As expected, Architectural sessions are the longest and the rarest. Note that there is some overlap: It is possible for a session to be characterized as both Masonry and either Painting or Decoration, for example. Restauration also overlaps with other characteristics. Inspecting these sessions further may reveal that they have phases in which one of the activity is prevalent. Next session describes the process we defined to support this activity.

Length	Blitz	Short	Average	Long	Marathon	Total
Spyware						
Architecture	0	1	5	15	11	32
Restoration	2	1	6	18	8	35
Masonry	26	43	41	76	5	191
Painting	51	39	31	40	6	167
Decoration	66	55	36	57	4	218
Project X						
Architecture	0	3	3	13	0	19
Restoration	0	9	14	21	0	44
Masonry	13	26	18	10	0	67
Painting	15	11	9	10	0	45
Decoration	9	4	3	1	0	17

Table 5.3: Session Types, for Project X and SpyWare

In general, we see that Project X has more Masonry and less Decoration session than SpyWare. A possible explaination for this is that Project X makes heavy use of a web framework, in which defining new classes of web components is commonplace. Project X also features more Restoration sessions, which can be possibly explained by the prototype, deadline-driven status of SpyWare. This also explains the presence of Marathon sessions in SpyWare.

5.4 Incremental Session Understanding

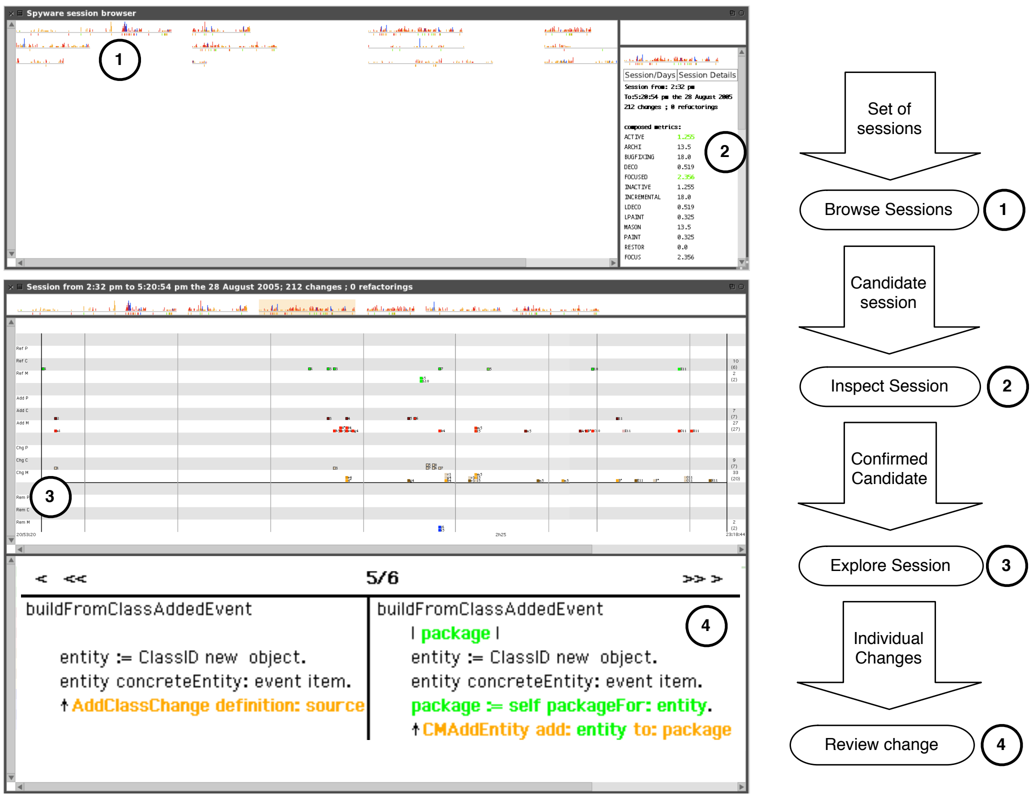
In this section, we outline the session exploration process we defined, before describing the tools and visualizations we have built to support it.

5.4.1 A Process for Incremental Session Understanding

We defined our session exploration process to allow the efficient navigation both between high-level changes (development sessions) and low-level changes (developer-level actions constituting a session).

Development sessions need to be summarized efficiently in a way which allows easy recognition of individual sessions. Upon closer inspection, key features and phases must be identifiable without involving too much cognitive effort. Developer-level actions need to be inspected closely, for actual program comprehension to take place. Navigating through related changes must be easy, and the extent of the changes must be assessed as quickly as possible.

To address these requirements, we defined a four step top-down session exploration process, starting with several development sessions, and ending with the inspection of individual changes. The process is shown in Figure 5.1. Initially, it takes as input a set of sessions of interest (for example, all the sessions in which a given class that one needs to understand



```
Session from: 2:32 pm
To:5:20:54 pm the 28 August 2005
212 changes ; 0 refactorings

composed metrics:
ACTIVE        1.255
ARCHI         13.5
BUGFIXING     18.0
DECO          0.519
FOCUSED       2.356
INACTIVE      1.255
INCREMENTAL   18.0
LDECO         0.519
LPAINT        0.325
MASON         13.5
PAINT         0.325
RESTOR        0.0
FOCUS         2.356
```

```
buildFromClassAddedEvent

    entity := ClassID new object.
    entity concreteEntity: event item.
    ↑AddClassChange definition: source
```

```
buildFromClassAddedEvent
    | package |
    entity := ClassID new object.
    entity concreteEntity: event item.
    package := self packageFor: entity.
    ↑CMAddEntity add: entity to: package
```

Figure 5.1: Session exploration and understanding process

was changed). Sessions are inspected as a whole, before individual sessions are selected for close review of their changes. The four steps of the process are:

1. **Browse Sessions.** In this step, one assesses a set of sessions all at once. The challenge is to summarize a large amount of changes (several sessions lasting several hours each) in a space compact enough that they can be encompassed at once, while retaining the ability to recognize individual sessions and gather superficial insight about their contents. We summarize an individual session in a *session sparkline*, which takes a very limited amount of screen space, allowing dozens of sessions to fit in a single screen.

2. **Inspect Session.** In this step, candidate sessions are inspected on their own. The session sparkline's interactive features are used, and phases in the session are recognized. The *session inspector* provides the values of metrics and the characteristics of each session in order to decide if a session should be inspected even further.

3. **Explore Session.** In this step, the actual understanding of the changes is supported by a detailed visualization of the changes in the session via the *session explorer*. The session explorer display changes emphasizing the entity they affect, their type (addition, modification, removal, refactoring action), and their change amount.

4. **Review Changes.** Finally, each individual change can be reviewed. This step includes actual code reading, which is eased via a *change reviewer*. The change reviewer highlights the actual change performed using a before/after view of the entity changed, and eases navigation to other changes of interest.

5.4.2 Browsing Sessions with the Session Sparkline

The first step of our process requires us to view several sessions at a glance. We do so with the help of an interactive visualization called the *session sparkline*. The session sparkline is influenced by Tufte's concept of the same name [Tuf06]. A sparkline is a word-sized graphic containing a high density of information. Figure 5.2 is an example of a session represented as a sparkline. The gray line in the middle of the figure is a time line. The default resolution of the figure is one pixel per minute (the example is magnified for clarity). Above and below the time line are bars representing the amount of changes occurring during a given interval (in our case a minute). Above the line are method-related changes: The height of these bars varies with the amount of change performed during the interval. Below the timeline are class-level changes. The class bars' height is constant as it is rare that two classes are changed in less than one minute. The color of each bar reflects the kind of change happening during the interval. A bar is orange if only modifications happened during that interval. It is red if at least one change is an entity addition, blue if one is a removal (superseding red), and green in case of a refactoring (superseding blue).

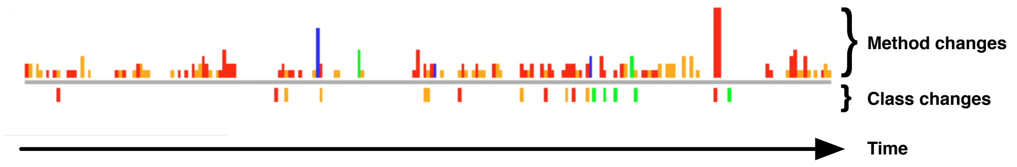


Figure 5.2: A session sparkline

The session sparkline sums up the activity pattern of a session at a glance, allowing one to immediately determine if a session has a lot of activity or not, and which kind of change dominates it. Activity patterns can be used to determine phases of the session. Assessing the length of a session is also immediate. Thus this representation ensures that each session has a distinct shape, making it easily recognizable across other sessions.

5.4.3 Inspecting and Characterizing Sessions with The Session Inspector

The session inspector's goal is to assist the interpretation of session sparklines in order to better characterize sessions. When summoned on a session, the inspector displays the various metrics we defined and their relative standing compared to the other sessions in the project. The inspector highlights the architectural, length, and activity characteristics that the session fulfills. It also lists the key entities of the session, *i.e.,* classes and methods that have been changed the most.

The interactive features of the session sparkline can be used during that interpretation phase: Hovering over any time interval will display a summary of the changes that were performed during that interval. The various phases of the session (separated by small periods of inactivity) can be inspected more closely to determine if they affect different parts of the system.

5.4.4 Viewing Changes in Context with The Session Explorer

The *session explorer* (Figure 5.3) support careful exploration of a session. It displays the exact nature of changes performed at a given point in time in a session. It acts as a portal between sessions and individual changes, as it is tightly integrated with the change reviewer, which assists program understanding.

Changes of the same type and on the same entity types are displayed on the same line, as squares. Change types are: modification, addition, removal and refactorings, while the entities considered are classes, packages and methods. The same colors than the sparkline are used, but get darker as the size of the change increases, to reflect the change amount. Each of these change figures has an identifier, so that changes applying to the same entities can be quickly related. They can also display a tooltip summing up the change as text. If clicked, a change reviewer is displayed for the given change.

5.4.5 Understanding Individual Changes with The Change Reviewer

The *change reviewer* eases the understanding of developer-level actions and the navigation between changes. The change reviewer displays a single developer-level action at once, using two panels (see Figure 5.1, panel 4).

Understanding incremental changes is eased by emphasizing them in before/after views of the entity. The view on the left shows the source code of the entity *before* the application of the change. It emphasizes removals of statements (in a red and struck-out font). The right view shows the source code of the entity *after* the change application. It emphasizes additions (in a green font) and renames (orange font) of entities. Further, since Change-based Software Evolution provides changes at the AST level, the changes are displayed at the level of AST entities, not lines. If a variable is renamed, only the variable will be highlighted, not the entire line, easing the localization of the change.

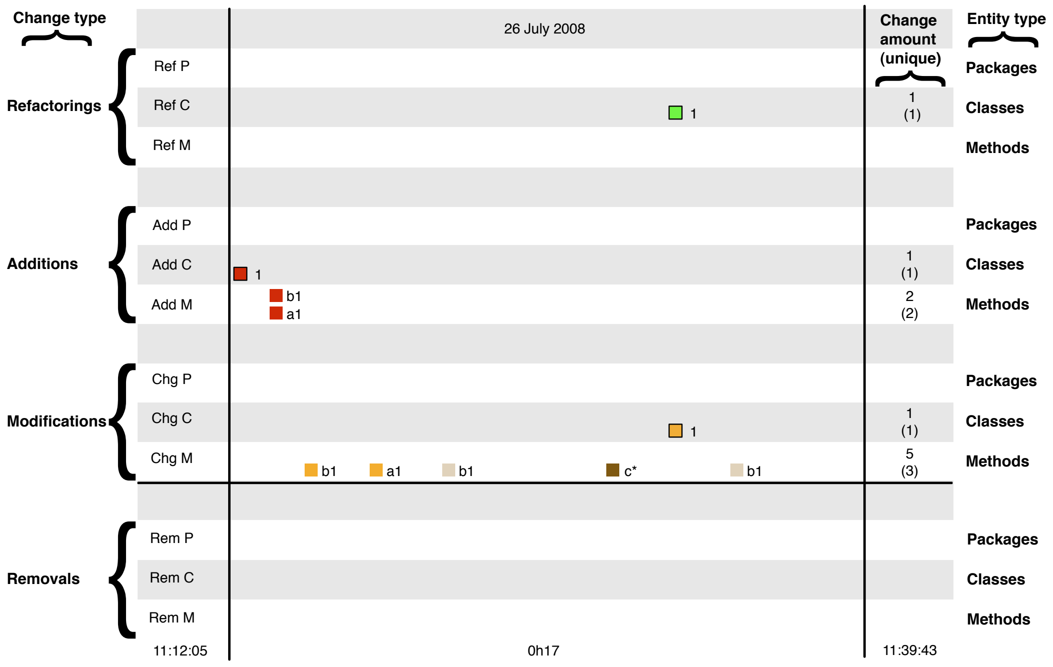


Figure 5.3: Overview of the session explorer

To ease navigation, the change reviewer offers shortcuts to related changes: The next/previous change to the same entity, the next/previous change in the session, and the next/previous session in the history.

In the remainder of this chapter, we report on our results without referring directly to the change reviewer in order to keep the discussion at a reasonably high level. Usage of the change reviewer is implied.

5.5 Validation

We now discuss selected example sessions in details. For each session we show the session sparkline, the primary characteristics, relevant metrics (with their value and their percentile in the project), key entities, and a snapshot of the session browser.

5.5.1 Decoration Session (Project X)

Sparkline:
Characteristics:
Decoration, Blitz
Metrics:
SA – 1.63 (90%) – Session Activity
ACC – 7 (74%) – Average Changes per Class
ACM – 2 (90%) – Average Change per Method
MCM – 4 (67%) – Most Changed Method
Key entities:
Methods *a* (`periodicalCallback:`) and *b* (`renderPeriodicalOn:`)

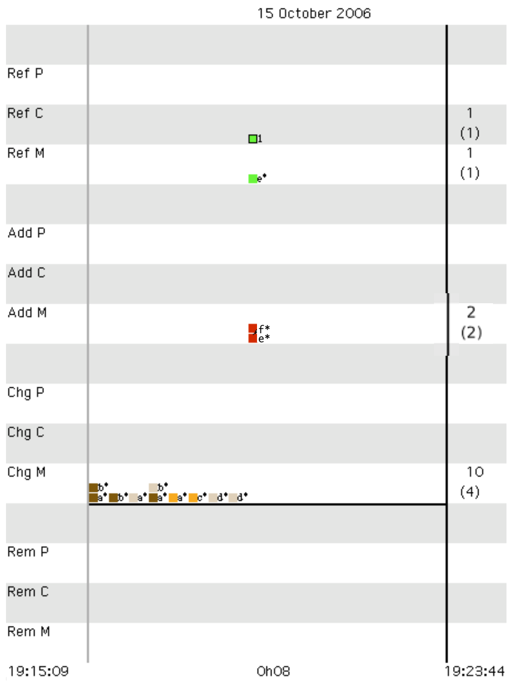


Figure 5.4: Decoration Session

Analysis This short session (8 minutes) consists mainly of decoration, *i.e.,* method modifications. It features towards the end a small amount of masonry and minor restoration (Figure 5.4). Its interesting characteristics are its high activity (1.6 changes per minute) and the first part of it where methods a and b are modified together several times (four times for a, three times for b, raising the *ACM* metric to 2), evoking high logical coupling. A look at the source code reveals that they are two HTML generation methods belonging to the same class. Methods c and d, and the methods e and f (created in the same minute, and bearing the same name, but on two different classes), are also related to HTML generation.

Conclusions We see that the link between methods a and b is emphasized by the sequential changes they were involved in. In the same fashion, the link between methods e and f is very strong, as they were created in the same minute. On the other hand, the refactoring changes were marked as such and could be reviewed faster. This is an example of session-based program understanding highlighting and prioritizing relationships between entities.

5.5.2 Painting Session (Project X)

Sparkline:
Characteristics:
Painting, Short
Metrics:
SA – 0.86 (63%) – Session Activity
ACC – 1.75 (11%) – Average Changes per Class
NIC – 12 (80%) – Number of Involved Classes
ACM – 1 (17%) – Average Changes per Method
NAM - 18 (63%) – Number of Added Methods
Key entities:
Several implementors of `filename`

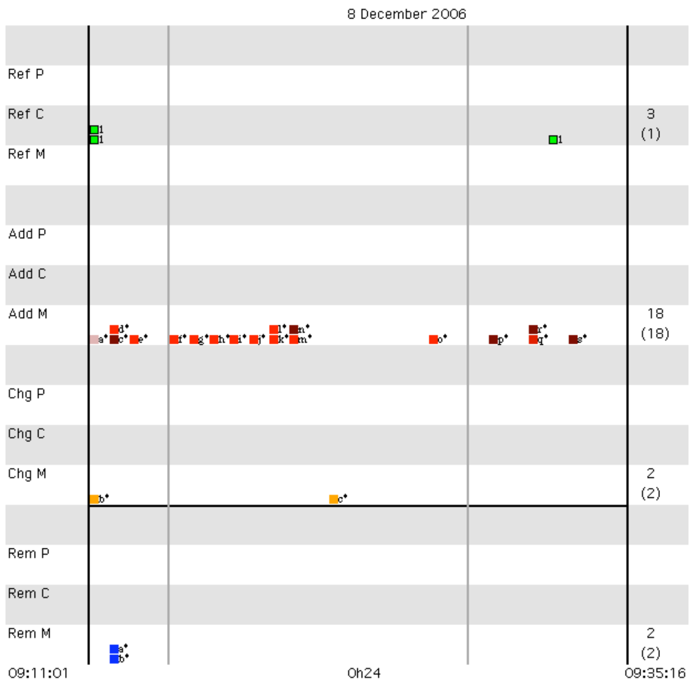


Figure 5.5: Painting Session

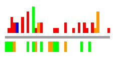
Analysis Figure 5.5 shows a peculiar session since its beginning shows the quick addition of methods to several classes. It is again quite short (25 minutes). The speed of the initial changes suggests that the task is repetitive. A closer inspection shows that the methods a, to m (excluding b and c) have the same name and are added to a hierarchy. They each return a constant, which explains why they are developed in succession. This explains the high values of NIC, NAM and the low values of ACC and ACM. These are characteristic of a crosscutting session. In that case it is justified by the protocol extension.

Once this is done, the rhythm slows down, and some actual logic is added to the system. This trend is started by method n, which specifies a test that needs to be fulfilled for the implementation to be correct. Later in the session, a strategy for file downloading is implemented relying on two possibilities. It is closely related to the first part of the session since methods a, d to m were referencing file names, used in methods b and c to build URLs.

Conclusions Reviewing this session with our approach emphasizes crosscutting changes. Instead of being spread out on several entities, the sequencing information allowed us to review the addition of methods to a class hierarchy (thus extending its protocol), in a sequential order. The ease of navigation between previous and successive changes made the connection more obvious. This is yet another example of session-based understanding highlighting relationships between entities.

From then on, understanding the remaining subset of changes –those having actual logic– was made simpler. The usage of the previously added protocol on the hierarchy was also obvious to relate to the later changes. This is an example of session-based understanding naturally dividing an implemented task into smaller, easily understandable subparts.

5.5.3 Masonry & Restoration Session (Project X)

Sparkline: 

Characteristics:
Masonry, Restoration, Short
Metrics:
NR – 14 (79%) – Number of Refactorings
SA – 1.53 (89%) – Session Activity
TNC – 58 (76%) – Total Number of Changes
ACC – 7.25 (77%) – Average Changes per Class
NCC – 6 (97%) – Number of Class Changes
NAM – 19 (80%) – Number of Added Methods
Key entities:
The entire `PRCommand` class hierarchy

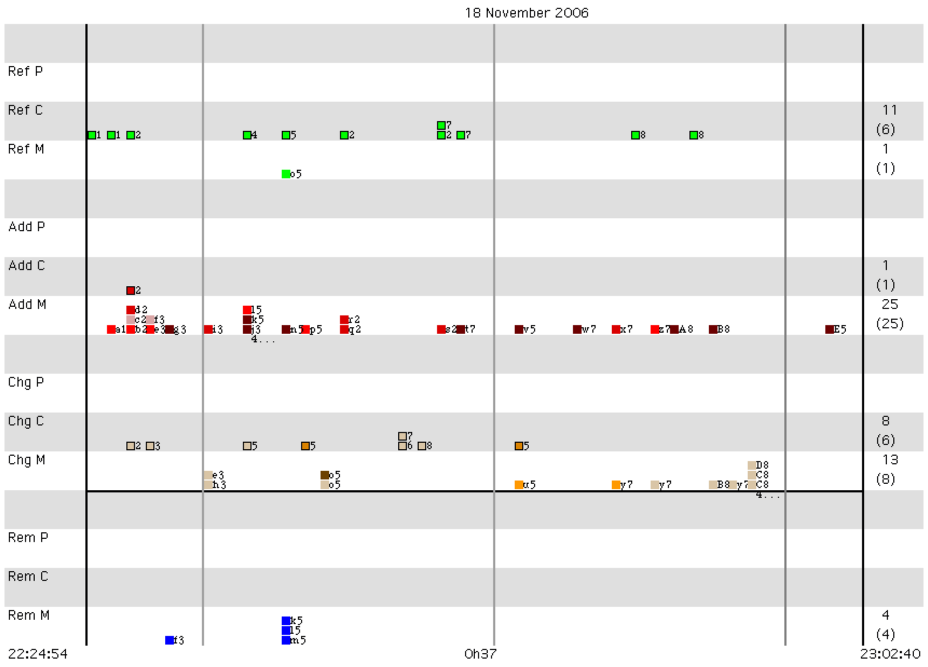


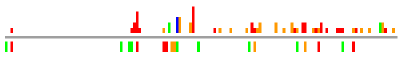
Figure 5.6: Masonry & Painting Session

Analysis Figure 5.6 shows an intense 37 minutes long masonry and painting session featuring a lot of class-based development. This is reflected in the metrics, featuring a high value for both ACC and NCC –which is unusual. One class is added –it is the focus of the session– while 8 class modifications happen during the first half of the session. Looking at the class modified and referenced in the session, we found that it is included in a hierarchy of classes following the *Command* design pattern [GHJV95]. The developer is fast at implementing the new command, which is actually a *Composite* Command, another design pattern. The methods added to this class show the minimal protocol expected from a member of the command hierarchy: `execute`, `validate` and `initialize`.

Afterwards, an extended protocol is added to other classes of the hierarchy with the methods `doAnswer` and `commitToCommands`. This session is a good example to follow, should the system need to be extended with a new kind of command by a less experienced developer. The characteristics "Masonry, Short, Active" seem to be good indicators of potential examples implemented by an experienced developer.

Conclusions Some characteristics of development sessions can be signs of a developer using domain knowledge to implement design patterns. Using our approach, these can be found and subsequently documented. A less knowledgeable programmer (new to the project or taking over a part of the system he does not know well) can use that example as an indication of what needs to be done when implementing a new instance of this domain-specific pattern. This domain-specific knowledge is important: Gamma *et al.* mention that design patterns are general solutions to problems, bound to be adapted to the specific requirements of every system [GHJV95].

5.5.4 Architecture & Restoration Session (SpyWare)

Sparkline: 
Characteristics:
Architecture, Restoration, Long
Metrics:
NR – 8 (94%) – Number of Refactorings
TNC – 80 (88%) – Total Number of Changes
NAC – 5 (95%) – Number of Added Classes
NIC – 16 (95%) – Number of Involved Classes
NAM – 30 (93%) – Number of Added Methods
NTM – 35 (89%) – Number of Touched Methods
Key entities:
`SWSession`, `SWQueryWrapper`, `SWChangeExplorer`,
`changeDescription`, `sessions`, `printAuthoredChange:`

Analysis We finish with a longer session (see Figure 5.7) from our own prototype, featuring architectural changes and restoration activities. This long session lasts for 2 hours and 25 minutes. During its implementation, the model of SpyWare was extended to include session-level changes, and a simple tool was implemented. This is reflected in the metrics, which show a very high amount of new behavior (NAC, NAM), across a large number of classes (NIC). From the sparkline we can divide the activity in 3 parts: **F1** shows nearly no sign of activity, **F2** is constituted of two activity spikes stopping at around half of the session, then **F3** finishes with a more stable output. We see that refactorings are applied consistently during the session, and that **F2** has a higher ratio of additions in its first spike. We now describe each part of the session in detail:

F1: F1 lays the ground work for the session by defining the `ChangeExplorer` class and changing its sister class, `ChangeExplorerTest`. A period of perceived inactivity ensues, which can be interpreted as either a design phase or a documentation phase. Since SpyWare was not able at that time to record navigation information, knowing more about the exact activity is not possible.

F2: The first spike adds a new element to the system: An interface centralizing queries to the model and its sister test class. The last methods in the first spike is a stub method called `sessions`, indicating the intention of using the session concept in the `ChangeExplorer` tool. A short period of inactivity follows, quickly replaced by the second spike of **F2**. In it, two classes are defined, the `ChangeGroup` and the `Session` class representing a session of changes. Several class modifications are made as `ChangeGroup` becomes a superclass of several classes in a large refactoring. Indeed, **F2** has most of the refactoring activities in the session, with some movements of functionality. Some methods are pushed up (restoration) and `ChangeGroup` becomes an abstract class, with `Session` and `Refactoring` inheriting from it. Alongside this, the `sessions` method is modified to exploit these new classes, as well as its test method.

F3: Once **F2** finishes, the architectural phase of the session slows down. The implementation of the actual tool is done in **F3** mainly using Painting. In class `SWSession`, the method **n5**, called `changeDescription`, is modified repeatedly. A closer analysis shows it returns a textual representation of a change, used in the `Explorer` tool. The end of the session adds a new class (11), called `ChangePrinter` and is then exclusively focused on it. `ChangePrinter` is in fact used in method D, modified just before the introduction of `ChangePrinter`. Looking at the code we notice that the class of method D is called `AuthoredChange`, and the method `changeDescription`. Looking at the code of the last methods, we deduce that `ChangePrinter` is a printing class introduced to handle the changes defined in `AuthoredChange`, using a double-dispatch mechanism close to the visitor pattern.

To sum up this session, we can discern and describe 5 phases: (1) a design/information gathering phase where development was slow, (2) the definition of the query interface and the definition of sessions, (3) the architectural changes to the model to add sessions, (4) the implementation of the `ChangeExplorer`, and (5) the implementation of a dedicated change printing subclass. Such an incremental vision of the session's history gives a clearer insight on the process than just considering the final outcome: 6 classes were added, 4 others were modified, 27 methods were added and 13 more were modified.

Conclusions This is another example of session-based program comprehension breaking down the understanding of a complex change to a sequence of smaller tasks. The long session was split in three parts, two of these being subsequently split again –showing the recursivity of the process. Restricting the change amount to consider at any given point makes individual changes easier to understand.

Attempting to understand a change of the same size without sequence information would be much more difficult. One could think changes of this size are quite rare, but they are not. In Table 2.1 (page 21), 25% of the commits spawned several files, and 2% spanned five files or more. This kind of activity happens frequently during active development of subsystems. It can also be due to developers committing seldomly, or submitting only patches when they have not access rights to a repository. In all these cases, incremental session understanding would be of help to understand the changes performed.

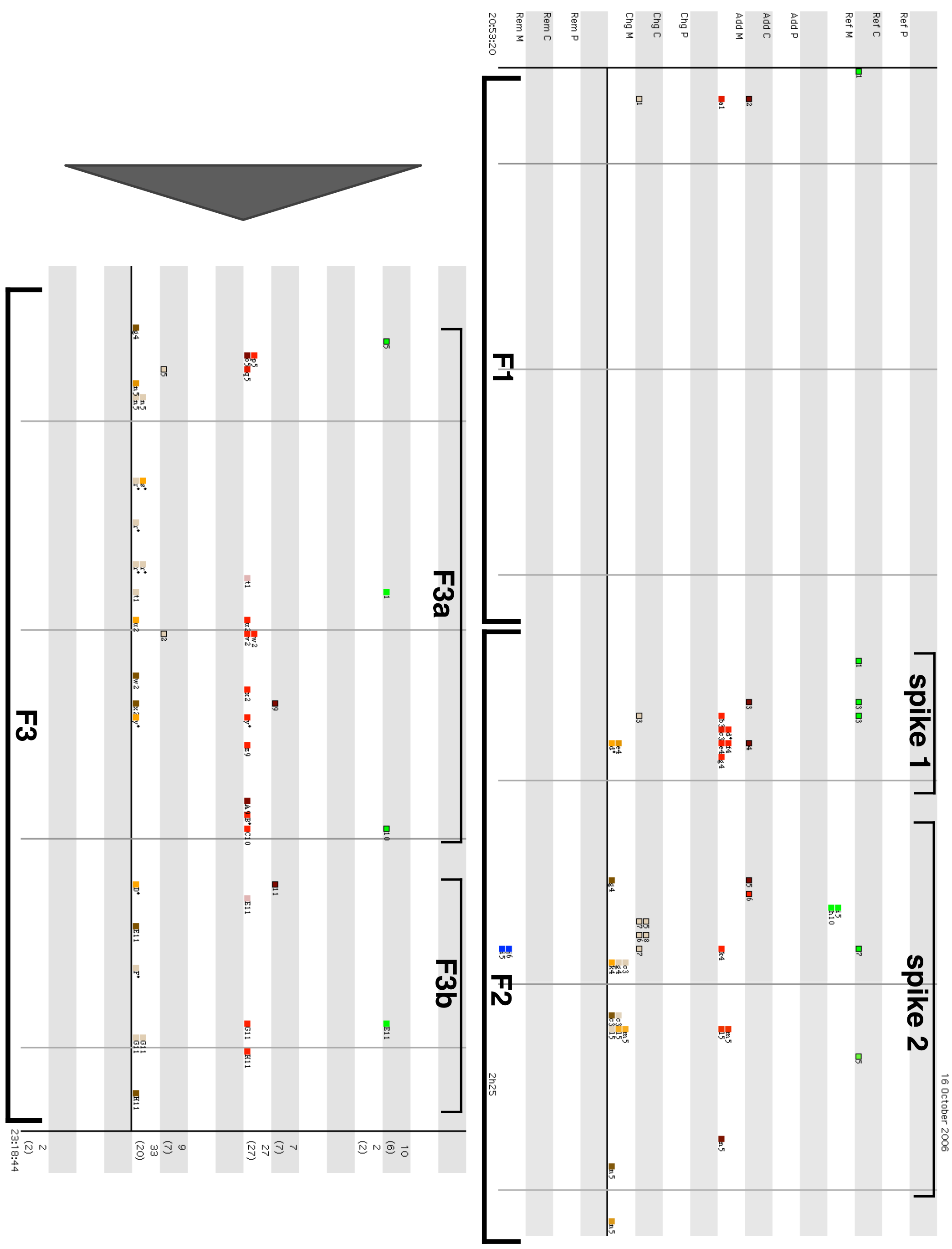


Figure 5.7: Session F: Architecture and Restoration

5.6 Discussion

Related Work

Related work in session-based program comprehension is non-existent, because no other approach records the data needed –or is able to recover it– with enough accuracy.

SCM system store only snapshot of the system, at intervals whose frequency is dictated by the developer. They thus contain only the outcome of a given task: All the incremental and sequence information is lost, which has the effect of blurring the relationships between entities, and forcing the programmer to understand the entire change at once.

On the other hand, approaches based on IDE monitoring keep the sequence information, but have a too shallow change representation to allow actual program representation. Their change representation is –at best– limited to knowing when an entity changed, but not how. Examples are Mylyn, by Kersten *et al.* [KM06], and work by Zou *et al.* [ZGH07]. Other IDE monitoring approaches store only navigation information, and totally bypass change information, such as NavTracks by Singer *et al.* [SES05], and work by Parnin *et al.* [PG06].

Parnin advocated merging both SCM and IDE monitoring [PG06], but combining the approaches would still lose the incrementality of the changes. To our knowledge, such an approach has not been implemented yet.

Smalltalk Change Lists

The closest data source to ours is the one found in Smalltalk change files, based on the same IDE notification mechanism our approach uses. Modification to classes and methods are stored in a text file for the primary usage of change recovery when the environment crashes. As a consequence, the tool support is limited to a simple chronological list of changes. Only versions are stored, and the changes themselves are not recovered: Displaying the differences between two versions is done on a line-by-line basis, which is harder to read than a syntax-aware differencing. Changes are also condensed in a single view, rather than two before and after views. To our knowledge, nobody used this data to perform program comprehension.

Dealing With Errors

One argument against incremental session understanding is that the recorded changes may contain errors that would have been corrected later on in the session, and would hence not appear in the SCM repository.

A possible way to deal with this issue is to mark certain entities as *transient*. A transient entity is a program element which is created and deleted in the same session. This is easy to determine with Change-based Software Evolution, since the change history of the entity will be entirely contained in one session.

Another area of future work is to locate actual bug fixes in the change history. Gîrba uses the assumption that a method which is only changed between two version of the system has had a bug fixed [Gîr05]. We hope Change-based Software Evolution will allow us to characterize bug fixes with other activity patterns, the primary one being Blitz Decoration sessions.

Phases of Sessions

Some of the sessions we reviewed had several phases that we identified visually. An automated approach to split a session in phases, or on the other hand, to link related sessions, would greatly assist incremental understanding.

Additional Information

Some additional information would help in understanding session. Recent versions of SpyWare record more than only changes: They also record navigation information (which method is viewed when), execution of code (and when an error occurs), and usage of the versioning system. Navigation information would give more context for understanding the changes, while code execution and errors would tell us whether a session is dealing with bug fixes or not. Finally, usage of the versioning system would tell us if our assumption that one session is equivalent to a commit is accurate.

Generalizability

We analyzed the development sessions across two projects, totaling more than 600 sessions. We can not however account for each type of project. In particular, the style of each developer vary greatly. This is why, for instance, we used relative thresholds for most of our metrics, so that sessions would be compared only with other sessions originating from the same project. Further studies are needed to set the thresholds to more empirical values.

For incremental understanding, we demonstrated its feasibility on several cases. More studies are of course needed, but the principles of incrementality were verified: In each case, reviewing the changes in sequence proved to assist program comprehension.

5.7 Summary

Program comprehension is a difficult task as it is unclear which path one needs to follow in order to understand a static piece of code. By recording the *exact sequence* of changes that took place when a given feature was implemented, Change-based Software Evolution conserves the logical path which the programmer took when building it. Following these so-called “programmer’s footsteps” while understanding a given piece of code is easier because of the following reasons:

- Related entities are changed together, even if no obvious link in the code exists. Examples of this are polymorphic methods (the method which ends up being called depends on the run-time type of the object, but the programmer knows which object is the most relevant to the task at hand), as in the last session we surveyed, or method communicating by means of side effects. The maintainer spends less effort querying the system to find the next entity to understand: He or she can focus on the actual understanding of the code.

- Changes are contextualized: It is easy to recognize that a change was performed in a refactoring, as evidenced in all the sessions we surveyed. Knowing this information allowed us to skim over these particular changes in order to focus on the changes which were *not* part of refactorings. A further context that eases understanding is obtained by the *session characterization* we introduced, which classifies sessions by their type, length and a variety of other metrics measuring various aspects of the session, such as the change amount and how crosscutting it was.

- Incremental understanding is supported. Instead of being confronted only the finished piece of code, the user can first review its initial versions, in which the general intent might not be hidden behind special cases that are bound to appear as time goes by. When the time comes to understand these changes, the newer changes defining a particular special case can be reviewed in priority. Our syntax-aware change viewer ensures that these changes are properly emphasized.

- Some repetitive patterns can be looked for and reused as examples. The third session we reviewed contained such a pattern as the developer reused his previous knowledge to efficiently implement a new feature according to the Command design pattern, adapted to its particular domain.

The characterization and the process we defined ease session-based program understanding as they support the navigation in and between sessions, augment the context by highlighting traits or characteristics relevant to the session, and ease the understanding of individual changes as we provide change-aware syntax highlighting of source code.

The information needed for incremental program understanding is only available through Change-based Software Evolution. Versioning system archives do not store the incremental process one took to build a given piece of code, only its final outcome. On the other hand, lightweight IDE monitoring tools store a change representation which is too shallow to reconstruct the actual incremental steps, if they store one at all.

We validated the effectiveness of session-based program characterization and understanding on four distinct examples, and from this conclude that program comprehension is significantly helped by Change-based Software Evolution. Change-based Software Evolution allows one to comprehend changes in an incremental fashion. The process and characterization we defined enables it at several levels, from high-level sessions to low-level individual changes, and to transition between levels fluidly.

Chapter 6

Measuring Evolution: The Case of Logical Coupling

Metrics are ubiquitous in software engineering, and especially in software evolution as a way to summarize large amounts of data. It is thus natural to evaluate how our Change-based Software Evolution can assist the definition, accuracy and usage of metrics.

How much are entities related to each other? Several metrics exist to answer this question. Logical coupling measures how often entities change together, and is a good measure to extract non-obvious relationships between entities: Entities might change together even if they do not reference each other or do so by indirect means.

Logical coupling has been traditionally computed based on transactions in a versioning systems, giving equal weight to all the entities modified in the same development session. With a more detailed change history, where the actual development sessions are recorded, we can recover relationships with more precision since a different weight can be given to entities changed in the same session.

6.1 Introduction

A significant portion of the reverse engineering field is dedicated to metrics and measurements. In Chapter 3, we have seen how Change-based Software Evolution can define system and change metrics. In Chapter 4, we used a fine-grained metric, the average size of methods in statements. Change-based Software Evolution eases these measurements since its system representation is very fine-grained. An approach based on an SCM system would first need to parse the entire system before providing this kind of metric. In Chapter 5, we used change metrics, most of them specific to our approach, to help us characterize sessions.

In this chapter, we continue this evaluation of Change-based Software Evolution for measurements. We evaluate how much the accurate system and evolutionary representation provided by Change-based Software Evolution improves the definition and the accuracy of metrics. We use the example of one of the most useful evolutionary software measurement, logical coupling.

Coupling was first used –alongside cohesion– as an indicator of good design by Parnas [Par72]. Parnas advocates that components in a software system should have a high cohesion and a low coupling to other components. If two components are highly coupled, chances are that changing one requires changing the other. Briand *et al.*, among others, correlated coupling between components with ripple effects [BWL99].

Coupling transitioned from being an indicator to an actual measurement, for which several metrics exist. Briand *et al.* gathered and formalized the variations between metrics in a comprehensive framework [BDW99]. A drawback of these metrics is that they require an accurate system representation, usually including calls between components and accesses to variables. Over the years, several alternative measures of coupling have been proposed, such as logical coupling [GHJ98], dynamic coupling [ABF04], or conceptual coupling [PM06].

Logical coupling is an evolutionary metric based on the change history of entities. The rationale behind logical coupling is that *“entities which have changed together in the past are bound to change together in the future”*. Logical coupling is computed from the versions of the system archived in a SCM such as CVS or Subversion [GJK03] (From now on, we refer to this measure of logical coupling as *SCM logical coupling*).

However, SCM logical coupling suffers from the inaccuracies of SCM systems. In this chapter, we investigate how much logical coupling can be improved by taking into account finer-grained change information. We compare SCM logical coupling against alternative logical coupling measurements in a prediction benchmark on two case studies with a large history.

Contributions. The contributions of this chapter are:

- Several novel measures of logical coupling using fine-grained change data. They take into account the amount and the precise time when the changes were performed.
- A comparative evaluation of these measures compared to SCM logical coupling on the change history of two case studies. The evaluation assesses how well the measures estimate logical coupling with less data.

Structure of the chapter. Section 6.2 explains the shortcomings of SCM logical coupling, describes its usages and the approaches that address its shortcomings. Section 6.3 details our evaluation procedure to measure the accuracy of the logical coupling measures. Section 6.4 describes the various alternatives to logical coupling we defined and report on their accuracy. Finally, Section 6.5 discusses the approach, while Section 6.6 concludes.

6.2 Logical Coupling

In this section, we first recall how logical coupling has been used for various activities, then explain the shortcoming of the current measure of logical coupling, before describing alternative measures in the literature.

6.2.1 Usages of Logical Coupling

Logical coupling is primarily used in reverse engineering as a means to detect dependencies between components. There are two reasons for this: (1) SCM logical coupling is cheaper to compute than a traditional coupling measurement such as the calls between components —as logical coupling does not involve parsing the entire system, but only the transaction logs— and (2) logical coupling can uncover *implicit dependencies*. Examples of implicit dependencies are indirect calls between classes, use of the reflective facilities in some languages, or code that communicates through side-effects. In all of these cases, a change to one of the classes requires the other class to be changed as well; this change is recorded in the versioning system, while it is not easily captured by program analysis.

Gall *et al.* first introduced the concept of logical coupling [GHJ98] to analyse the dependencies in 20 releases of a telecommunications switching system. The concept was soon adopted by other researchers in the context of reverse engineering and program comprehension. Biem *et al.* defined visualizations to recognize change proneness, and defined an aggregated measure of all the change coupling relationships of a class [BAY03]. Pinzger used logical coupling as part of his Archview methodology [Pin05] for architecture reconstruction. D'Ambros visualized logical coupling with an interactive visualization called the Evolution Radar [DL06a].

Logical coupling has also been used for change prediction. Zimmermann *et al.* [ZWDZ04] presented an approach based on data mining in which co-change patterns between entities are used to suggest relevant changes in the IDE, when one entity in the relationship is changed by the programmer. Ying *et al.*, and Sayyad-Shirabad *et al.* employed a similar approaches [YMNCC04; SLM03], although at a coarser granularity level. These approaches suggests files, while Zimmermann's employs lightweight parsing to recommend finer-grained entities.

6.2.2 Shortcomings of SCM Logical Coupling

Computing logical coupling based on an SCM system is restricted by the two shortcomings of SCM we identified in Chapter 2, information loss and coarseness. Figure 6.1 shows a hypothetical development sessions explaining these shortcomings.

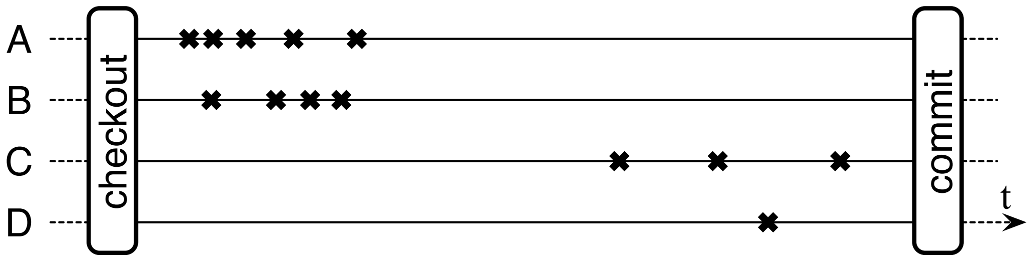


Figure 6.1: A development session involving four entities

SCM Logical coupling suffers from information loss. In the figure, entities A, B, C and D, are modified during a single development session. The figure shows a timeline for each entity, with a mark every time the entity was changed during the session. It is obvious that entities A and B have a very strong relationship, while entities C and D have a moderate relationship. In addition, the relationships AC, BC, AD or BD, are weak at best. However, based only on the information recovered from the version repository, an SCM-based logical coupling algorithm will give equal values to each relationship. This means that *a large amount of data is needed before the measure can be accurate*. The threshold commonly used to establish a strong logical coupling between entities is 5 co-change occurrences [GJK03]. Zimmermann *et al.* found that change prediction works much better for projects with a large history, in "maintenance mode", such as Gcc, rather than in active development [ZWDZ04].

In addition, SCM systems are file-based. Without additional preprocessing like the one employed by Zimmermann *et al.*, the relationships will be computed only at the file level. Knowing the relationships at a finer level, such as methods, is useful in a reverse engineering context. There is a difference between a coupling involving most of the methods in two classes, and one involving only a handful of them: The second one is much more tractable.

6.2.3 Alternatives to SCM Logical Coupling

Given the shortcomings of Logical Coupling, two alternative measurements have been proposed to improve its accuracy.

Zou, Godfrey and Hassan introduced Interaction Coupling [ZGH07] to address information loss. Interaction coupling is based on IDE monitoring, like our approach, and records two types of events during development sessions: Navigation events and edition events. These events are counted at the file level, and the number of *context switches* between two files is the measure of interaction coupling. The measures are also classified in three categories: co-change (the two files changed at least once together), change-view (one of the files changed, while the other was consulted) and co-view (the two files were viewed together). Only the sequence of events is taken into account, not their date or their contents.

Fluri *et al.* developed an approach to identify and classify changes between versions, beyond file-level changes [FG06]. Each type of change has a significance ranging from *low* to *crucial*. Taking into account the significance of changes when computing change coupling gives a different measure of it. The comparison with logical coupling was performed on a case-by-case basis. Moreover, Fluri's approach still relies on standard versioning systems.

Change-based Software Evolution shares some of the advantages of Fluri's approach. In particular, our AST-based changes automatically filter out layout changes, or changes to comments. We can also filter out changes performed by refactoring tools (which are behavior preserving), as we monitor the tools themselves.

6.3 SCM Logical Coupling Prediction

Since we define alternative measurements to logical coupling, we need a way to evaluate them. The criteria we chose is the approximation by alternative metrics of future SCM Logical Coupling with a shorter history. This section explains our evaluation strategy.

6.3.1 Motivation

The amount of data needed by logical coupling is one of the reasons logical coupling is used more for retrospective analysis, rather than forward engineering. In plain words, if there is not enough data, the measure is useless. For instance, Zou *et al.* mention in their study that SCM logical coupling was unable to find any coupling relationship from a one-month period of data [ZGH07]. On the other hand, the coupling they defined did work on shorter periods than the classic logical coupling. Their comparative evaluation of the two metrics was however anecdotal. Since we have recorded a larger amount of data, we can compare more formally the accuracy of the coupling measures, by employing a predictive approach.

6.3.2 Procedure

We first need to gather the predictions of the approaches, and the actual events they predict.

To gather the set of actual strong relationships in the system, we measure the SCM logical coupling in the entire system. We then select the relationships which have a logical coupling beyond the threshold value used by Gall *et al.* [GHJ98], which is 5. This constitutes the expected set of strongly coupled entities E.

For each logical coupling measure m we measure the coupling of each relationships for each session. This coupling is between 0 and 1. If a relationship's coupling is above a certain threshold tr, we put the relationship in a candidate set Cm_{tr}. The relationships where an entity has changed less than 5 times overall are filtered out, since we can not verify the prediction for these. To be extensive, we tried threshold values between 0.01 and 0.99, with a 0.01 increment.

To evaluate the impact of adding data, we repeat this procedure for two or three sessions, *i.e.,*, the threshold for a relationship has to be crossed in two (respectively three) sessions in the history to add an entity in the candidate set.

6.3.3 Evaluation

We evaluate the accuracy of the measures by comparing the candidate sets with the expected set in terms of precision and recall. We define the precision P and recall R for a candidate set C, with respect to the expected set E, as:

$$P = \frac{|E \cap C|}{|C|} \qquad \text{and} \qquad R = \frac{|E \cap C|}{|E|}$$

Precision and recall come from information retrieval and give an idea of the accuracy of a prediction [vR79]. The recall expresses the number of false negatives given by the measure: It is the proportion of expected entities that were predicted. If all the expected entities are predicted, the recall is 1. The precision evaluates the number of false positives in the prediction: It is the proportion of predicted entities that were wrong. If only entities in the expected set are predicted, the precision is 1.

Intuitively, using two or three sessions instead of one should decrease recall and increase precision. Increasing the threshold also increases the precision and decreases the recall. The relationship is however not linear.

To more formally elect the best thresholds and the best coupling measures, we combine precision and recall into the F-measure, defined as their weighted harmonic mean:

$$F_\beta = \frac{(1+\beta^2) \cdot P \cdot R}{\beta^2 \cdot P + R}$$

Common variations from $\beta = 1$ give a stronger weight to precision ($\beta = 0.5$), or to recall ($\beta = 2$).

6.3.4 Result Format

For each measurement, we present its accuracy in our prediction benchmark with two precision/recall scatterplots, one for each case study. Each point represents a threshold value (between 0.01 and 0.99), its x-coordinate being the recall of the measurement for the threshold, and its y-coordinate being the precision. Both vary between 0 and 1.

We also provide a table with the best possible f-measures for each case study, taking into account each parameter: The number of sessions, and the weight to give to F.

6.3.5 Data Corpus

We compared the coupling measures over the change histories of our two projects with the most coupled relationships according to our previous definition.

The first of those is SpyWare. For this study we selected the first three years of development of SpyWare, representing 500 sessions of development for a total of 500 class relationships marked as highly coupled.

The second project is Software Animator, a system written in Java over 134 sessions and a period of three months, obtained via the Eclipse version of our plugin. It features around 250 class relationships considered as highly coupled.

6.4 Logical Coupling Measurements and Results

In this section, we present the coupling measurements we defined or reproduced. For each measure, we note which aspect of the evolution the measure emphasizes, explain its intuitive meaning, give the actual formula we use to compute it, and the accuracy results we found.

For any program entity a and session s, we note δ_a for any change concerning a or the children of a (changes to a method also concern its class) and $s_a = \{\delta_a \in s\}$. We compute the coupling $a \leftrightsquigarrow b$ between two entities by aggregating a per-session coupling measure over a set of sessions. The various coupling measures each define their own $\overset{s}{\leftrightsquigarrow}$.

6.4.1 SCM Logical Coupling

Emphasizes Occurrence of co-change of two entities during the same sessions.

Intuition This logical coupling measurement is the one introduced by Gall *et al.* Two entities are related if they change during the same session. A threshold of five co-change occurrences is often used to qualify entity as logically coupled.

Definition

$$a \overset{LC}{\leftrightsquigarrow} b \overset{def}{=} \begin{cases} 1 & \text{if } a \text{ and } b \text{ changed during } s; \\ 0 & \text{otherwise.} \end{cases}$$

Results Table **??** presents the F-measures of SCM Logical Coupling. This measurement serves as the baseline for our further measurements. Since this is the measure we try to predict, it will make every recommendation that will eventually reach 5 co-change occurrences. Hence, its recall is always 1. However, its precision is very low, yielding very low values of F for one and two sessions. Of course, the more sessions are taken into account, the more accurate the measure becomes.

6.4.2 Change-based Coupling

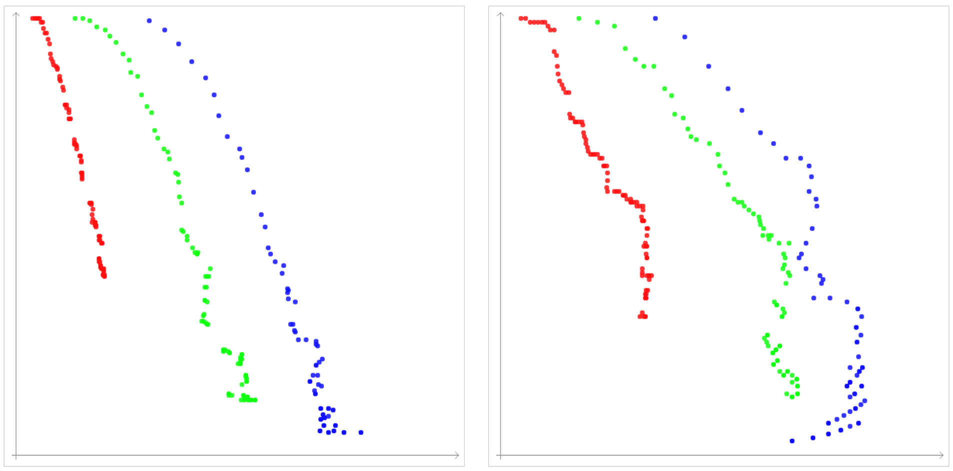
Emphasizes How much an entity changed during a development session.

Intuition Entities that change many times during a session are more coupled than those which only change occasionally. This is similar to the LC measure except the number of changes for each entity is factored into the measure.

Definition

$$a \overset{CC}{\leftrightsquigarrow} b \overset{def}{=} \left(\prod_{s_a \times s_b} |s_a| \cdot |sb| \right)^{1/|s_a \times s_b|}$$

Results Figure 6.2 is the precision-recall graph on the two case studies. F-measures are shown in Table 6.1. This measure is a significant improvement over the baseline, especially when only one or two sessions of information are taken into account. This measure performs best with medium (for higher precision, or with less sessions) to low (for better recall, or with more sessions) thresholds.



(a) SpyWare (b) Software Animator

Figure 6.2: Graphs of Precision (X axis) and Recall (Y axis) of Change Coupling: 1 session (red), 2 sessions (green), 3 sessions (blue)

6.4.3 Interaction Coupling

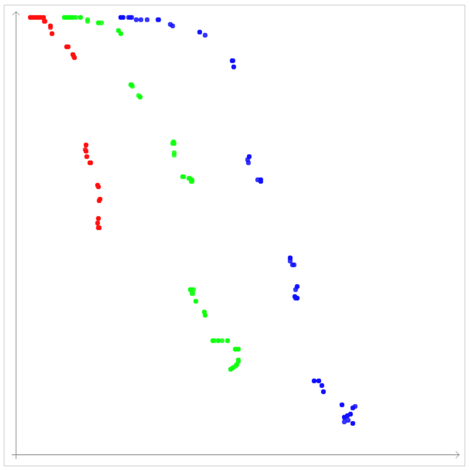
Emphasizes Interleaving of sequential changes.

Intuition This measure is related to the one introduced by Zou *et al.*, although we consider only the code changes and ignore the navigation events. Each time an entity changes, it becomes the entity in focus. The coupling between A and B is equal to the number of times the focus switched from A to B or from B to A. The original version of the measure is then rounded between zero and one, based on whether the number of context switches is below or above the historical average. To keep the accuracy of the measure, we do not round it.

Definition

$$a \overset{IC}{\leftrightsquigarrow} b \overset{def}{=} |s_a \times s_b| \quad \text{with } \delta_a \text{ and } \delta_b \text{ successive}$$

Results Figure 6.3 is the precision-recall graph on the two case studies. F-measures are shown in Table 6.1. If all measures improved the accuracy of Logical Coupling by comparable amounts, this measure is the best performing. When not rounded, it has better results, albeit by a low margin. As for the Change Coupling, medium to low thresholds work best.


(a) SpyWare

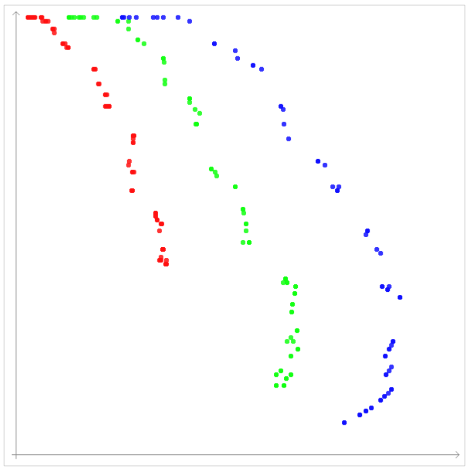

(b) Software Animator

Figure 6.3: Graphs of Precision (X axis) and Recall (Y axis) of Interaction Coupling: 1 session (red), 2 sessions (green), 3 sessions (blue)

6.4.4 Time-based Coupling

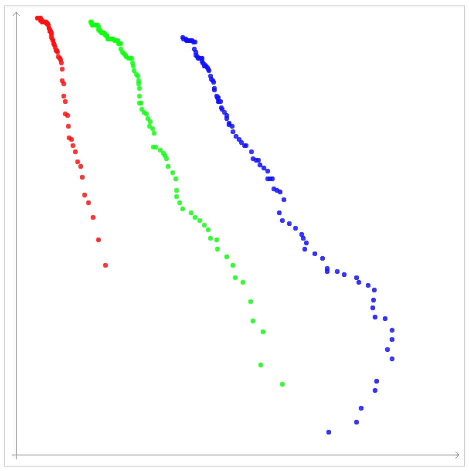
Emphasizes Proximity in time of changes in a session.

Intuition If two entities changed simultaneously, their relationship is stronger than if one changed at the beginning of the session and the other at the end. The coupling linearly decreases with the average delay between changes, from 1 if all changes happened simultaneously to 0 if it is one hour or more.

Definition

$$a \overset{TC}{\leftrightsquigarrow} b \overset{def}{=} \max\left(0,\ 1 - \frac{1}{|s_a \times s_b|} \sum_{s_a \times s_b} |\Delta t(\delta_a, \delta_b)|\right)$$

Results Figure 6.4 is the precision-recall graph on the two case studies. F-measures are shown in Table 6.1. Unlike the previous two measurements, Time Coupling works best with high to medium thresholds. It performs slightly worse than other measures for low number of sessions, but performs closer for a higher number of sessions.


(a) SpyWare

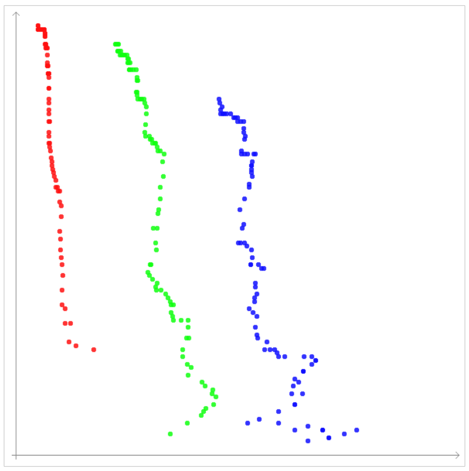

(b) Software Animator

Figure 6.4: Graphs of Precision (X axis) and Recall (Y axis) of Time Coupling: 1 session (red), 2 sessions (green), 3 sessions (blue)

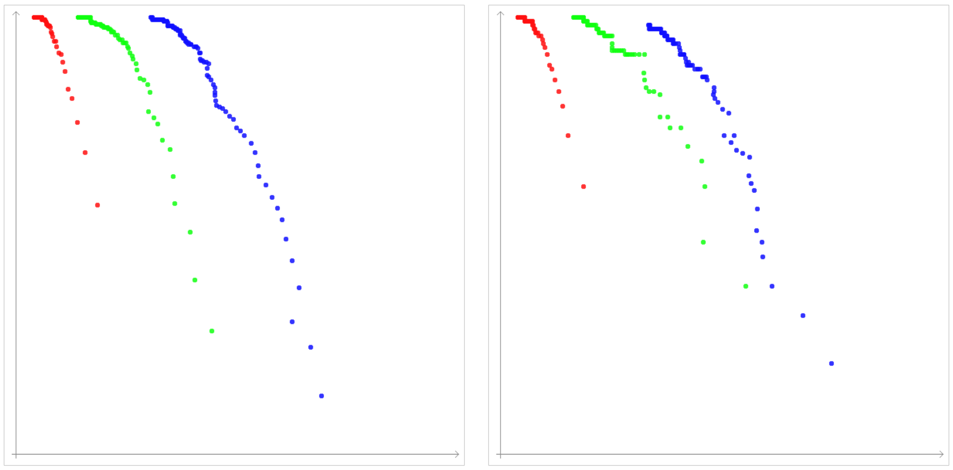
6.4.5 Close Time-based Coupling

Emphasizes Close proximity in time of changes in a session.

Intuition If two entities are usually changed close together, but one experiences changes much later in the session, their relationship will decrease. To counter this, this coupling averages only the five lowest time intervals.

Definition The definition is identical to the Time Coupling, however only the five lowest time intervals are kept.

Results Figure 6.5 is the precision-recall graph on the two case studies. F-measures are shown in Table 6.1. This measurement performs better than the regular Time Coupling, but needs very high thresholds. This is to be expected, since it averages the smallest time intervals, hence providing higher values on average.



(a) SpyWare (b) Software Animator

Figure 6.5: Graphs of Precision (X axis) and Recall (Y axis) of Close Time Coupling: 1 session (red), 2 sessions (green), 3 sessions (blue)

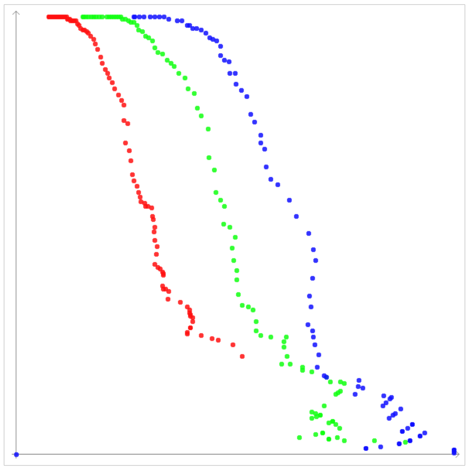
6.4.6 Combined Coupling

Emphasizes All the previous characteristics.

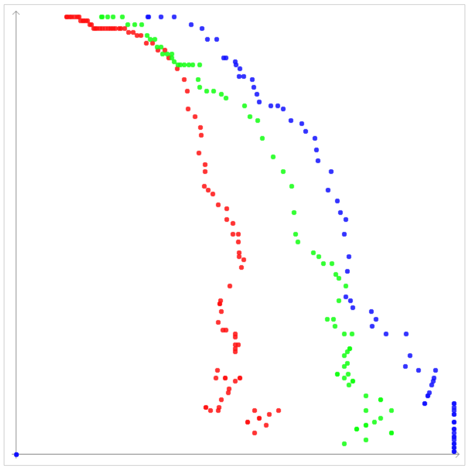
Intuition All the coupling definitions we described yielded an improvement over the default definition. By combining the three measurements, we may have an even better measure.

Definition Average of the normalized values of the Change Coupling, the Time Coupling, and the Interaction Coupling. Time Coupling was selected over Close Time Coupling since the optimal thresholds are closer to the ones of the two other measures.

Results Figure 6.6 is the precision-recall graph on the two case studies. F-measures are shown in Table 6.1. If there was no clear winner in the previous measurements –save the Interaction Coupling by a small margin– combining the measures yields a significant improvement. This improvement is best in the most important case, for one or two sessions. The F-measure for one session is actually comparable with the one of the other approaches, but for two sessions. The same is valid for two versus three sessions. For three sessions, its accuracy is in range with the other measurements.



(a) SpyWare



(b) Software Animator

Figure 6.6: Graphs of Precision (X axis) and Recall (Y axis) of Combined Coupling: 1 session (red), 2 sessions (green), 3 sessions (blue)

	SpyWare			Sw Animator		
	1 Session	2 Sessions	3 Sessions	1 Session	2 Sessions	3 Sessions
SCM Logical Coupling						
$F_{0.5}$	3.9	6.1	14.1	3.4	5.4	12.5
F	13.5	20.0	38.4	14.7	21.6	40.8
F_2	28.1	38.5	61.0	28.6	39.1	61.6
Change Coupling						
$F_{0.5}$	22.5	28.1	39.7	36.2	41.4	49.6
F	44.0	47.0	61.0	61.5	57.7	68.0
F_2	56.6	59.2	72.2	69.6	68.3	76.3
Interaction Coupling						
$F_{0.5}$	22.0	28.8	42.2	36.0	40.9	54.4
F	43.3	49.0	60.1	56.5	55.0	67.7
F_2	58.1	63.9	77.4	72.1	68.8	79.1
Time Coupling						
$F_{0.5}$	22.8	27.8	38.8	18.7	20.4	30.0
F	48.8	47.5	61.0	37.7	45.4	61.1
F_2	66.4	61.0	74.8	57.0	61.9	70.8
Close Time Coupling						
$F_{0.5}$	21.6	28.1	41.3	22.0	29.0	42.4
F	41.5	46.8	62.2	51.4	54.6	67.6
F_2	59.3	61.3	74.6	58.9	62.6	74.5
Combined Coupling						
$F_{0.5}$	41.5	39.9	55.1	50.7	53.9	68.9
F	49.9	55.3	69.0	63.8	64.0	72.7
F_2	62.7	64.1	77.9	70.3	70.2	78.2

Table 6.1: Best F-measures for all logical coupling measurements

6.4.7 Discussion of the Results

The measures perform comparably on both case studies, except for the general tendency to perform slightly better on our second case study. We are not sure of the cause of this behavior. Possible reasons could be differences in style of programming, designs of the system, or programming language used (one was built in Smalltalk, the other in Java). Increasing the data used in the benchmark would help us see the overall trend better.

However, all measures performed more or less comparably across the two case studies, and most of all, performed significantly better than SCM Logical Coupling with a lesser, but more detailed, history. This gives us confidence that our alternative measures of logical coupling can be used faster than the previous one.

Finally, combining the metrics works surprisingly well, especially in the cases where the history is the most limited, with only one or two sessions in which a co-change event happened. We think that there is still room for improvement, since our definition of the Combined Coupling is a simple average. One possible way to improve this would be to give different weights to the metrics, or change the way the average is computed to give more weight to high metric values.

6.5 Discussion

Recording

In the absence of explicit commits, SpyWare automatically starts a new session whenever there is a gap of one hour or more between changes. We make the assumption that the session boundaries is where commits would occur.

Precision

When more accurate information is taken into account, the logical coupling is more stable, and can thus be used earlier on to make predictions. SCM logical coupling is often used for retrospective analyses when the history is considerable. We provided initial evidence that more detailed measures provide useful results earlier.

Generalizability

Our experiment was carried on a small sample (500 and 250 strong coupling relationships). We can not generalize it to other systems. However, the measures performed comparably on both case studies. Further, all of them performed, in both cases, with a significant improvement over the default measure of logical coupling. Once we gather more case studies, we intend to replicate the experiment. In the meantime, the size of the improvement makes us confident that our results will be verified.

Replication of Fluri's Approach

We did not attempt to replicate the coupling measured by Fluri *et al.* [FG06] for the following reasons: (1) Some of the changes in Fluri's taxonomy are Java specific, and could not be translated to Smalltalk, and (2) the measure of change coupling integrating significance was specified informally.

Replication of Zou's Approach

On the other hand, we partially replicated the interaction coupling of Zou *et al.*. One limitation is that we consider only the change events, not the navigation events, since our prototype did not record them over the whole period. One improvement over the original approach is that we do not round the measure at the end of each session. The original metric return a binary metric for each development session, depending on whether the number of context switches was greater than the average. Instead, we return a more precise value between 0 and 1. We then use this value in combination with a threshold to evaluate its accuracy.

Method-level Coupling

We also performed a preliminary classification of the coupling relationships between classes based on how the methods in the classes were related. We can detect if a coupling is caused by a large number of methods (which is less manageable), and whether these methods call each other directly or not (which makes the coupling harder to detect). However these results are not mature enough to be discussed further.

6.6 Summary

In this chapter, we evaluated how Change-based Software Evolution can improve the accuracy and the level of detail taken into account in software measurements. CBSE maintains at all time an accurate, AST based representation of evolving systems, down to individual statements and the entities (classes, methods, variables) they represent. This ensures that measures computed on top of data provided by Change-based Software Evolution are accurate since the finest level of detail can be used if necessary.

In addition, CBSE closely monitors the evolution of systems by recording, instead of recovering changes. This considerably increases the accuracy of evolutionary metrics, as well as the accuracy with which we can follow the evolution of more static metrics.

We demonstrated the improvements provided by these two aspects on logical coupling. While a coarse measurement of logical coupling yields boolean values at the file level, measurements defined on Change-based Software Evolution data yields values in a range (from zero to one), on finer-grained entities such as classes and methods.

The consequence of these improvements is that less evolutionary data is needed for the measurements to be reliable. Where logical coupling was used primarily for retrospective analyses –as a long system history was needed–, our measurements can be used sooner and hence support active development. We verified this by defining a prediction benchmark and concluded that 1 or 2 occurrences of co-change could quite accurately predict whether future co-change occurrences would take place. Several metrics were shown to have good predictive powers, around 50% in both precision and recall with only two sessions of data instead of 5. In addition, combining metrics increased the accuracy further.

This shows that the fine-grained data provided by Change-based Software Evolution is still useful when summarized at a very high-level by measurements, as these are more accurate than measurements based on coarser data.

Part III

How First-Class Changes Support Software Evolution

Executive Summary

This part of the dissertation shows how, beyond understanding systems, Change-based Software Evolution can assist programmers to actually perform new changes. We show that:

CBSE helps to automatize repetitive changes. *In* **Chapter 7***, we show how we extended CBSE with program transformations behaving as change generators. This has two advantages. The first is that the transformations are fully integrated in the system's evolution, and can be further used for program comprehension, or to ease the transformation's evolution. The second is that concrete recorded changes can be refined into generic transformations, providing a concrete bases for transformation definition.*

CBSE improves recommender systems. *In* **Chapter 8***, we show how CBSE can be used to define a benchmark for code completion. In* **Chapter 9***, we show how CBSE improves on the existing, SCM-based benchmarks for change prediction. In both cases, CBSE-based benchmarks allow to reliably and repeatedly evaluate several variations of recommender algorithms. In both cases, the best performing algorithms use fine-grained change data to make their predictions.*

This shows that recording fine-grained history has a lot of diverse usages beyond the obvious uses in reverse engineering. We expect more of these usages to be defined in the future.

Chapter 7

Program Transformation and Evolution

When a system needs repetitive changes, programmers are faced with a choice: Either perform the change manually, running the risk of introducing errors in the process, or use a program transformation language to automate the task. We tackle three problems related to program transformations, and their integration in Change-based Software Evolution.

- *We first extend our change metamodel to support parametrized program transformations in a natural fashion.*
- *Second, we propose an example-based program transformation approach: Instead of using a transformation language, recent changes to the system are used as concrete examples which are generalized to define program transformations.*
- *Finally, we show how program transformations can be integrated in the overall evolution of a system, and the possibilities this enables.*

7.1 Introduction

Program transformations automatize repetitive changes that would be error-prone if performed manually. Program transformations are ubiquitous: They span a broad spectrum from compilers transforming high-level programs to machine code, up to refactorings, available in every IDE. It seems natural to investigate how well Change-based Software Evolution can support and interact with program transformations. We decompose the interplay between Change-based Software Evolution and program transformations in three problems:

Transformation support: How can Change-based Software Evolution be extended to define generic program transformations, and to which degree extending our model to support program transformations is natural.

Transformation definition: How can recorded manual changes be used to ease the definition of program transformations, by making more explicit the process through which transformations are created from concrete examples.

Transformation integration and evolution: How can program transformations be integrated in our vision of an accurate description of a program's evolution, and what are the consequences of this integration.

Each of these problems is related to a different aspect of Change-based Software Evolution.

The first problem is spawned by the fact that the changes we defined are already program transformations, albeit basic ones. In Chapter 3, we specified that each change is executable and affects an AST. Changes are in essence constant transformations. We want to see how far our model can be extended to support parameterized program transformations without degrading it.

The second problem stems from our previous observations that recording changes gives us considerably more information than is available in an SCM system. We want to investigate to which extent the structure and the order of recorded changes is useful to express program transformations. We call this approach *example-based program transformation*.

Finally, the last problem is related to our desire to model evolution with accuracy. We already model and record a subset of program transformations, namely refactorings. We used these for reverse engineering and program comprehension purposes in Chapter 5. In the same fashion, we investigate how we can document when and where a program transformation was applied to the system. In addition, we explain how documenting transformation applications could assist in transformation maintenance, automation and evolution.

Contributions. The contributions of this chapter are:

- An extension to our change metamodel to define parametrized program transformations, which views program transformations as change generators.
- An example-based program transformation approach to assist the definition of program transformations. It is based on (1) the recording of a sequence of changes to provide the initial transformation structure to be worked on, (2) a direct interaction with this structure to refine and generalize it, and (3) the interaction with example entities to set the scope of the newly defined transformation.
- A proof-of-concept of Example-based Program Transformation through three distinct examples that demonstrate its versatility: (1) Flexible refactoring definition, such as the "extract method with holes" refactoring, (2) Program-specific code transformations, exemplified by replicating changes to code clones in a code base, and (3) Definition of "informal aspects", exemplified via the definition of a simple logging aspect.
- How to fully integrate program transformations in a system's evolution, and a description of the consequences of transformation integration for their comprehension and evolution.

Structure of the chapter. We first describe how we extend our change metamodel to include program transformations in Section 7.2. In Section 7.3 we motivate and outline our approach to define transformations from examples found in the history. We describe its steps in detail in Section 7.4, illustrated on a running example. We describe additional examples in Section 7.5. In Section 7.6, we describe transformation integration and evolution. Finally, we discuss our approach in Section 7.7, and conclude in Section 7.8.

7.2 Change-based Program Transformations

The definition of a program transformation is simple. It is a function which takes as input a program and a set of parameters, and returns a modified program. Our changes fit that definition, except that they do not accept parameters: Each change encodes a constant transformation. In this section we extend our model with generic changes, which support parameters. We first describe variables and their roles, then generic atomic changes, transformation application, and finally change-based control structures.

7.2.1 Variables And Their Roles

In our model, program transformations are sequences of generic composite changes or generic atomic changes. Each of these changes reference several *variables*. When the transformation is applied to a system, each variable will be eventually bound to the ID of a given program element. Depending on the natures of the references to it in the atomic changes of a program transformation, a variable can have three roles:

Constant: The variable is involved in a creation change. Its ID is guaranteed to be generated at instantiation time. No further treatment is necessary.

Parameter: The variable is not created in the change. Its ID will be given to the transformation as an argument as it is instantiated.

Unlocated: The variable is not created in the change. Rather than being given as a parameter, it is computed from other parameters and the state of the system the program transformation is applied to.

From this, we observe that only the variables which have the role of *parameters* need to be assigned an ID when the transformation is instantiated. Other variables will be automatically bound to the ID they need.

Heuristics are used to differentiate between parameters and unlocated variables. Variables in *parent* or *entity* slots of changes are preferably parameters. Variables in *location* slots are preferably unlocated. The roles are not fixed and can be changed afterwards.

7.2.2 Generic Changes

Generic atomic changes fields contain *variables*, instead of IDs of concrete entities. Whenever a variable is assigned an ID, all references to it in all the changes in the transformation are updated. When a transformation is applied to a system, each generic change in it is instantiated: It generates a concrete change by assigning IDs present in the system to variables. Executing the changes on the system modifies it according to the transformation.

All types of atomic changes have a generic counterpart, their behavior during instantiation is as follows:

Creation: Generates a new ID for its *entity* each time it is instantiated.

Addition/Removal: The *parent* or the *entity* must have an ID assigned, or the change fails. As an alternative, the *parent* or the *entity* can be computed via functions.

Insertion/Deletion: Works the same as an addition for *parent* and *entity*. The *location* inside the parent is determined by variables as well. Their ID must be known, or computed from the *parent*. In the latter case, the *location* has to be computed according to the state of the parent (*i.e.,* the contents of its AST) at instantiation time.

Property Change: The property *value* can be computed via a function.

7.2.3 Instantiation and Application of Transformations

Applying a transformation on a set of parameters works as follows:

Bind parameters to their actual values (IDs).

Instantiate each change. IDs of constants are generated. IDs in unlocated variables are computed as well. Should such a computation not succeed for any reason, the change fails.

Execute the changes. After each atomic change is instantiated, apply it directly to the code base. Changes later on in the transformation may depend on previous changes being applied.

Tag each concrete change generated by the transformation as being issued by the transformation. This is described in Section 7.6.

During instantiation, a list of all the already applied concrete changes for each generic composed change is kept. If a change fails, all the concrete changes generated by the generic composed change that have been executed so far are undone. The generic composed change then also fails, triggering a sequence of undo at the next level, until the entire transformation is undone.

7.2.4 Control Structures

To define more complex transformations, changes need to be applied differently depending on the parameter that is given. Consider for instance the case of a change that should be applied to all methods in a class, or only to methods whose name start with "test". For this we need control structure such as for loops and conditionals. Since our model supports composition, control structures are represented as special kinds of changes, that wrap one or more generic composite changes.

Iteration We allow a generic composite change (or more) to be applied to a set of entities via a generic *iteration change*. The collection of parameters to which the contained changes will be applied is computed by a function of other parameters and is called the *iteration set*. For instance, an iteration can take as parameter the ID of a class, and compute its iteration set as being all methods of the class. The wrapped change is then instantiated multiple times, once for each method.

Another kind of iteration consists in attempting a change an unspecified number of times, until it fails to apply. The iteration change intercepts the failure so that only the last application of the wrapped change is undone. A use case for this would be to replace all references in the system from one variable name to another.

Conditional Conditional changes wrap several generic composed changes. Each one is an alternative. When the conditional is instantiated, it instantiates each wrapped change until one does not fail. Previous failing changes are undone. If all changes inside it fail, the conditional itself fails. Optional changes are similar. They attempt to apply the changes inside them, but they do not fail if every wrapped change fails.

Calling Transformations Transformations can call one another. The calling transformation specifies the values of the parameters to give to the callee. This allows reuse of commonly used transformations as building blocks of bigger ones.

7.2.5 Wrap-up

Our change model is easily extended to implement transformations. By simply considering them as change generators, we added a layer above our previous layer which does not interfere with the layer below.

In addition, composition of changes is naturally extended to implement higher-level control structures such as iterations and conditionals, which control how the changes they encapsulate are instantiated. Even if the control structures we defined in such a way are simple, they have been sufficient so far. Our model can support the definition of more complex transformations in this way.

The remainder of this chapter deals with how recorded changes ease the definition of transformation as a sequence of changes, and the definition of the computations that take place in them.

7.3 Transforming programs by examples

If program transformations are useful, defining them is not always easy. It is often the realms of specialists: Compiler writers, Refactoring implementors, or users of program transformation languages. If a program transformation is outside that realm, such as a domain-specific transformation not large enough to warrant the use of a full-blown program transformation language, it will often end up being done manually, which is error-prone.

As an alternative to manual editing we present an *example-based* program transformation approach: To specify how a repetitive task should be automated, a programmer *records* a sequence of changes as an example of it.

The rationale behind our approach is that highly abstract activities such as defining a program transformation have less overhead when one is working on concrete instances of the problem. In one experience report of the DMS program transformation system [ABM$^+$05], the authors mention that before defining a large-scale transformation to be applied on several modules of a system, they first converted one module of the application by hand.

Since programmers need to work on concrete instances of a problem before defining transformation, our approach maximizes the usage of these concrete example. Starting from a recorded example working in its particular context, the developer generalizes it to make it applicable in other contexts. During this process, we allow the developer to directly interact with the structure of the transformation and the entities affected by the transformation. Finally, the programmer can explicitly name and store the newly defined program transformation, and reuse it as needed.

We first compare existing approaches to draw requirements for a transformation approach filling the gap that exists in the program transformation spectrum. We then give a bird's eye view of how we define a program transformation based on example changes, and show how our approach fulfills the requirements.

7.3.1 The Program Transformation Spectrum

Program transformation has been tackled in 4 areas [1]:

1. **Refactorings** [Fow02] are by now well integrated in many IDEs and –generally being one right-click away– easy to apply. They are also safe due to their behavior-preserving nature. They are part of many a developer's toolbox. They are however limited in scope: Only the handful of most common refactorings are available in IDEs. Implementing a new refactoring involves a significant coding effort: An example on the Eclipse website [2]. The refactoring described in the example is implemented in several Java source code files and is more than a thousand lines of code long, making such a practice out of reach for most users.

2. **Linked Editing** refers to the ability of some code editors [TBG04; DER07] to change a code fragment and have the editor broadcast the changes to similar regions of code, called clones. The clones can be either documented or detected by the tool. Since they usually work at the text level, not the syntactic level, their applicability is usually limited to code fragments with a high degree of similarity. Parameterizing an edit is not supported.

3. **Aspect-Oriented Programming** (AOP) allows crosscutting concerns to be abstracted and separated from the code base into aspects [KLM$^+$97]. As part of the compilation process, the program is transformed to include the aspects which were extracted. Using Aspect-Oriented Programming involves learning a new language, with all the hurdles that implies.

4. **Program Transformation Languages** are the most flexible and powerful approach, but the most difficult to successfully use. Transformations tend to deal directly with the AST of the program, whereas AOP uses special purpose (and more limited) constructs such as advices and pointcuts. Such languages are seldom integrated in a development process, but defined externally and applied to the entire program as a separate step of the build process. All these factors make the use of program transformation languages worthwhile only for large-scale, system-wide, transformations such as migrating code from one distribution framework to another [ABM$^+$05; RB04].

[1] Compilers are out of scope in this work.
[2] www.eclipse.org/articles/Article-Unleashing-the-Power-of-Refactoring/

In Table 7.1 we compare the approaches on flexibility, scale of usage, ease of use and IDE integration. From this, we extract the following requirements to ease transformation definition:

	Refactoring	Linked Editing	AOP	Transf. Languages
Flexibility	-	-	+	++
Transformation size	-	-	+	++
Ease of use	++	+	-	–
IDE integration	++	+	-	–

Table 7.1: Advantages and drawbacks of approaches in automated program transformation

IDE integration tremendously lowers the barrier to entry as the functionality is directly available.

Flexibility. Low flexibility rules out many smaller transformation tasks. To fill the spectrum between easy, but limited usage (refactoring, linked editing), and complex, large-scale usage (AOP, transformation languages), we need a sufficiently expressive and flexible approach, integrated in the IDE.

Low Abstraction Level. The flexibility offered by program transformation tools requires a high abstraction capacity, reducing the efficiency of most programmers. Even transformation experts need concrete examples [ABM+05]. A key requirement is to lower the abstraction level of the task, by giving it concrete foundations. The steep learning curve of program transformation languages (and to a lesser extent AOP) is due to the highly abstract nature of the tasks they involve: The programmer has to build a mental representation of the program as a data structure and manipulate it, without having an easy way to check the results.

We now describe Example-based Program Transformation in general terms before discussing how it fulfill these requirements.

7.3.2 Example-based Program Transformation in a Nutshell

Example-based program transformations use *recorded changes* as examples, refined into general-purpose program transformations. Defining and using example-based program transformations is divided in 6 steps. We describe these steps alongside a running example (in italics): The definition of an informal logging aspect. A full-fledged aspect would probably not be implemented as such in a project not already using AOP. Instead, developers might implement it manually by inserting the same instructions over and over in the source code, leading to maintainability problems in the long run.

Step 1: Record changes. To start the procees, one first needs to record a concrete example of a transformation. The transformation is manually performed on example entities. *The example change for a logging aspect is to introduce a logging statement at the beginning of a method.*

Step 2: Generalize changes in a transformation. This process is performed automatically given a concrete sequence of changes. Each reference to an entity ID in the change is converted to a variable. Based on how each entity is created, modified or removed in the change sequence, the system deduces a role for it. Some will be parameters to the transformation (i.e., specified before running it), while others will need to be computed from these parameters. *From the example change, the generalization process deduces that the change applies to a parameter, X, which is a method. It also deduces that the location where the statement is inserted is variable, and must be specified by the user.*

Step 3: Edit variables part of the transformation. Based on the roles of the variables deduced from the previous step, the developer edits the transformation and specifies how the values of the variables are computed. *The developer specifies the location inside X where to insert the statement, and also that the string printed in the logging statement should contain the name of X.*

Step 4: Compose changes. The developer can introduce higher-level constructs such as iterations or conditionals to build larger changes from elementary building blocks. *The developer specifies that to apply the logging transformation to a class, the previous change must be applied to all the methods of the class. He can also define variants of the change, depending on the number of arguments in the calling method.*

Step 5: Test the transformation on examples. At any time during steps 3 and 4, the developer can test the effects of the modified transformation by running it repeatedly on the example entities, to assess if the results match his needs on the initial example. *The developer can compare the results of the initial change and the specified transformation on the same targets.*

Step 6: Apply the transformation to the system. Once the transformation is defined, it is saved and can be immediately used from the code browser of the IDE. This allows the transformation to be applied to one entity at a time. A special-purpose tool allow the transformation to be applied to a larger number of entities if it is needed. *The logging transformation is stored, ready to be applied at any moment to any program. The transformation can also be undone.*

7.3.3 Does our approach fulfill the requirements?

IDE Integration. An IDE plugin monitors programmer activity, and records it as change operations. This is done silently, without interrupting the workflow of the developer. The subsequent refinement of the transformation is done using a user interface which still belongs to the IDE. Then, the transformation can be quickly accessed and tested on program elements since the tool has access to the program representation through the IDE.

Low abstraction. We kick-start the transformation process by extracting the initial transformation structure from the recorded example. The tool infers which parts of the transformation need to be further edited or not, giving the developer a concrete list of tasks to perform. The reified program transformation also allows direct interaction with the structure of the transformation. Parts of it can be easily edited, swapped, removed or cloned. The process to follow is given by the recorded changes in the transformation itself.

Flexibility. A transformation is not limited to a single entity, since an arbitrary number of changes can be recorded. It can also be edited to include higher-level control structures, such as iterations of a change on multiple entities, or trying alternative changes depending on the type of the entity a transformation is applied to. Unlike refactorings, we do not focus exclusively on behavior-preserving program transformations.

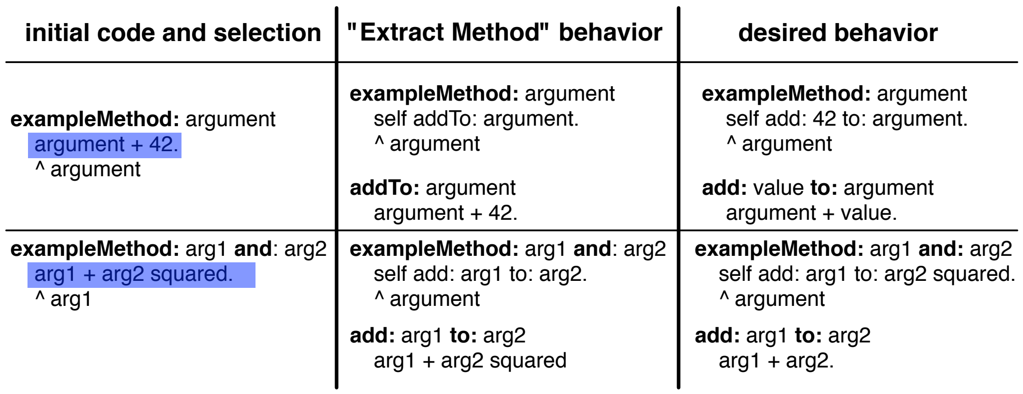
7.3.4 Running example

We present a more complex example, that we use in the following sections. It is an extension to the "Extract Method" refactoring. According to Fowler, "Extract Method" is the *Refactoring Rubicon*, i.e., a refactoring tool featuring "Extract Method" is probably complex enough to implement most refactorings[3]. Refactoring tools featuring "Extract Method" are able to infer which local variables need to be passed as arguments to the extracted block of code (those that are referenced both in the code block that is being extracted, and outside of it).

A frequent situation however is that a constant expression in the block of code would need to be passed as a parameter to the new method that is being created. Since this expression is referenced only inside the code block that is extracted, it is not converted to a parameter (See Figure 7.1, top). Another related situation is when a method call is used on an extracted variable. Usually, the call becomes part of the extracted code, while sometimes it should stay in the calling method (Figure 7.1, bottom). In both cases, additional modifications are needed: There are two possible alternatives, shown in Table 7.2.

Both approaches require several steps and disrupt the flow of the programmer. In the following we show how, using our approach, we create the "Extract Method with Holes" refactoring, *i.e.,* additionally to extracting a method, portions of constant code can be also extracted as parameters of the newly created method.

[3]See www.martinfowler.com/articles/refactoringRubicon.html



initial code and selection	"Extract Method" behavior	desired behavior

exampleMethod: argument
  argument + 42.
  ^ argument
	exampleMethod: argument
	  self addTo: argument.
	  ^ argument
	
	addTo: argument
	  argument + 42.
		exampleMethod: argument
		  self add: 42 to: argument.
		  ^ argument
		
		add: value **to:** argument
		  argument + value.

exampleMethod: arg1 **and**: arg2
  arg1 + arg2 squared.
  ^ arg1
	exampleMethod: arg1 **and**: arg2
	  self add: arg1 to: arg2.
	  ^ argument
	
	add: arg1 **to:** arg2
	  arg1 + arg2 squared
		exampleMethod: arg1 **and:** arg2
		  self add: arg1 to: arg2 squared.
		  ^ argument
		
		add: arg1 **to:** arg2
		  arg1 + arg2.

Figure 7.1: Actual vs expected behavior of extract method

A. Extract Temporaries	**B. Add Parameters**
1. Extract the constant expression to a temporary variable.	1. Extract the code of the method.
2. Move temporary variable declaration out of code block	2. Apply the “Add Parameter” refactoring to the newly created method.
3. Extract code block to new method	3. Replace constant expression with the parameter in the body of the newly created method.
4. Inline the temporary again.	4. Edit call site; add constant expression in place of new parameter.

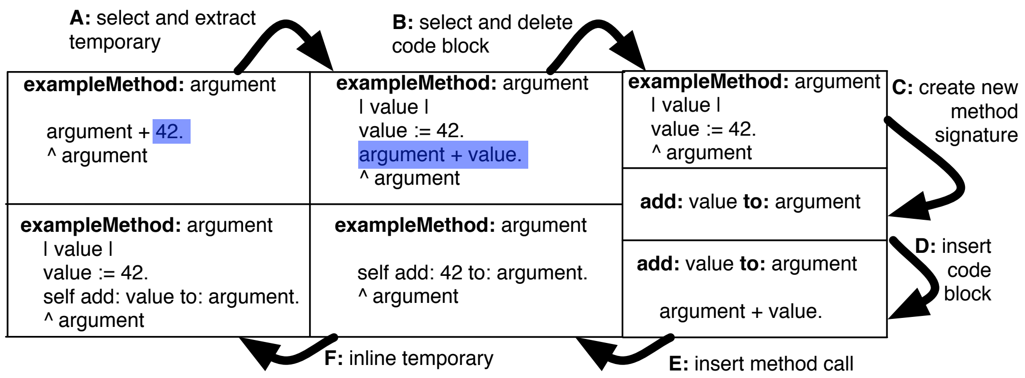
Table 7.2: Refactoring alternatives

7.4 The Six-step Program to Transformation Definition

7.4.1 Recording the example

Change recording has been discussed at length previously and is no different than the process described in Chapter 3. This step is fully automatic.

Example. For the "Extract Method with Holes" refactoring, six composite changes are recorded, as shown in Figure 7.2.


A: select and extract temporary
B: select and delete code block
C: create new method signature
D: insert code block
E: insert method call
F: inline temporary

```
exampleMethod: argument

    argument + 42.
    ^ argument
```

```
exampleMethod: argument
    | value |
    value := 42.
    argument + value.
    ^ argument
```

```
exampleMethod: argument
    | value |
    value := 42.
    ^ argument
```

```
add: value to: argument
```

```
add: value to: argument

    argument + value.
```

```
exampleMethod: argument

    self add: 42 to: argument.
    ^ argument
```

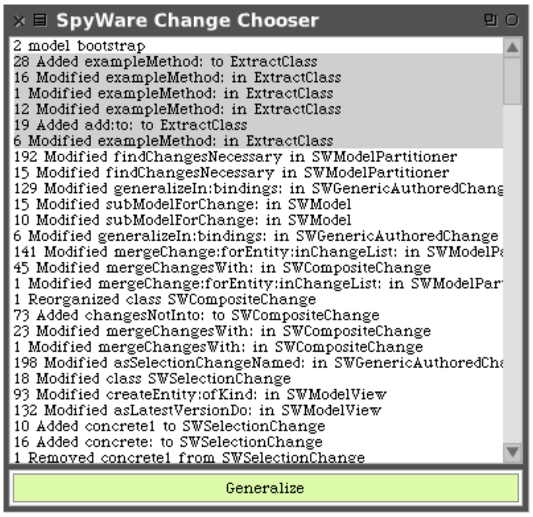
```
exampleMethod: argument
    | value |
    value := 42.
    self add: value to: argument.
    ^ argument
```

Figure 7.2: Recorded changes

7.4.2 Generalizing the example

To generalize an example, the developer looks into the change history, where the recent changes are stored, and selects the changes of interest, using the Change Chooser tool (Figure 7.3). This tool allows a developer to look for changes farther in history, or to unselect parts of the changes if other activities were performed in parallel that do not belong to the envisioned transformation.

Deducing the role of each variable. Given the structure of a generic change, a role (parameter, unlocated or constant) is automatically deduced for each variable, affecting how the programmer has to process it. Since changes can be composed, a given variable can play different roles in several parts of the change, e.g., it might be a parameter in some of the composite changes and a constant in another. At the transformation level, it will be a constant.



```
2 model bootstrap
28 Added exampleMethod: to ExtractClass
16 Modified exampleMethod: in ExtractClass
1 Modified exampleMethod: in ExtractClass
12 Modified exampleMethod: in ExtractClass
19 Added add:to: to ExtractClass
6 Modified exampleMethod: in ExtractClass
192 Modified findChangesNecessary in SWModelPartitioner
15 Modified findChangesNecessary in SWModelPartitioner
129 Modified generalizeIn:bindings: in SWGenericAuthoredChang
15 Modified subModelForChange: in SWModel
10 Modified subModelForChange: in SWModel
6 Modified generalizeIn:bindings: in SWGenericAuthoredChange
141 Modified mergeChange:forEntity:inChangeList: in SWModelPa
45 Modified mergeChangesWith: in SWCompositeChange
1 Modified mergeChange:forEntity:inChangeList: in SWModelPar
1 Reorganized class SWCompositeChange
73 Added changesNotInto: to SWCompositeChange
23 Modified mergeChangesWith: in SWCompositeChange
1 Modified mergeChangesWith: in SWCompositeChange
198 Modified asSelectionChangeNamed: in SWGenericAuthoredCha
18 Modified class SWSelectionChange
93 Modified createEntity:ofKind: in SWModelView
132 Modified asLatestVersionDo: in SWModelView
10 Added concrete1 to SWSelectionChange
16 Added concrete: to SWSelectionChange
1 Removed concrete1 from SWSelectionChange
```

Figure 7.3: The Change Chooser shows the recent changes to the system.

Example. Looking at the structure of the sequence of changes shown in Figure 7.2, the changes are generalized in the following way (roles are in italics):

Change A: The variable representing the number 42 is a *parameter*, which the user will set up via selection (see Section 7.4.3). The inserted statements (variable declaration and variable assignment), are *unlocated*.

Change B: The deleted block of code will also be a *parameter* of the transformation. Nodes under it are *constants*.

Change C: There are no parameters or variables needing a location, as every entity is created on the spot. They are *constants*.

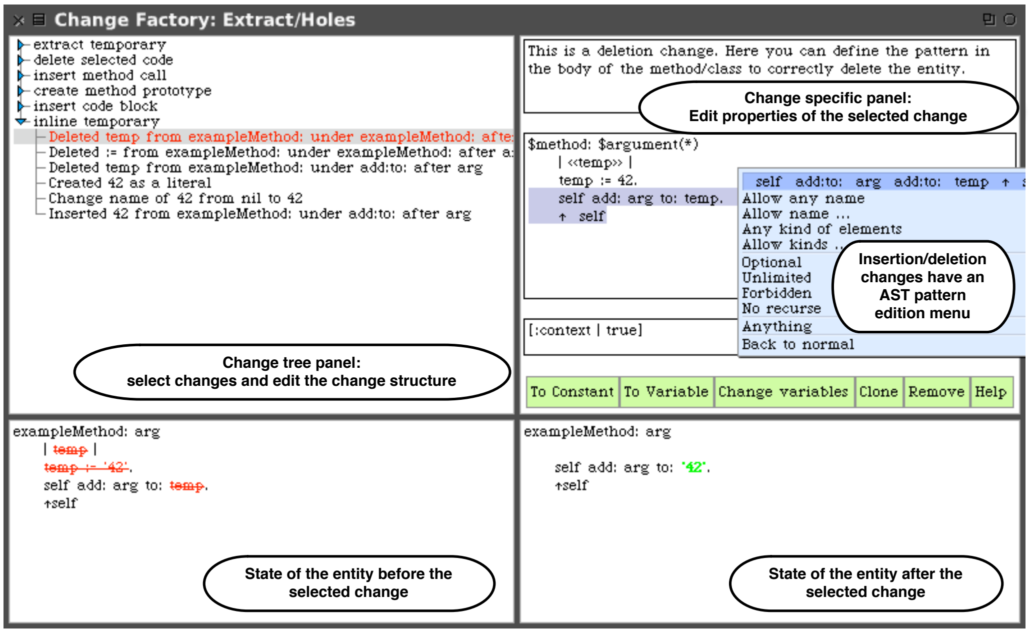
Change D: The block of code is *unlocated* and needs a location (the very beginning of a method). The *constant* method in change **C** is a *parameter* in change **D**.

Change E: The *unlocated* method call needs a location (the previous position of the selected code block).

Change F: The deleted statements (variable declaration, assignment and reference) are *unlocated*, while the value in the assignment (also *unlocated*) needs to be relocated where the variable reference was.

In the overall transformation, the only *parameter* is the method to which the refactoring is applied.

7.4.3 Editing the Example


Change Factory: Extract/Holes
Change specific panel: Edit properties of the selected change
Change tree panel: select changes and edit the change structure
Insertion/deletion changes have an AST pattern edition menu
State of the entity before the selected change
State of the entity after the selected change
To Constant
To Variable
Change variables
Clone
Remove
Help
self add:to: arg add:to: temp ↑
Allow any name
Allow name ...
Any kind of elements
Allow kinds ...
Optional
Unlimited
Forbidden
No recurse
Anything
Back to normal

```
extract temporary
delete selected code
insert method call
create method prototype
insert code block
inline temporary
    Deleted temp from exampleMethod: under exampleMethod: afte
    Deleted := from exampleMethod: under exampleMethod: after a
    Deleted temp from exampleMethod: under add:to: after arg
    Created 42 as a literal
    Change name of 42 from nil to 42
    Inserted 42 from exampleMethod: under add:to: after arg
```

This is a deletion change. Here you can define the pattern in the body of the method/class to correctly delete the entity.

```
$method: $argument(*)
    | «temp» |
    temp := 42.
    self add: arg to: temp.
    ↑ self
```

```
[:context | true]
```

```
exampleMethod: arg
    | ~~temp~~ |
    ~~temp := '42'~~.
    self add: arg to: ~~temp~~.
    ↑self
```

```
exampleMethod: arg

    self add: arg to: '42'.
    ↑self
```

Figure 7.4: The Change Factory's main interface, shown editing a deletion change

Editing the example is the first step requiring manual work. Figure 7.4 shows the Change Factory, the prototype tool in charge of editing and testing program transformations. Using this tool, one can:

- Change properties of variables such as their name.
- Specify the location of unlocated variables via user selection or pattern matching.
- Specify conditions which may prevent or modify the action of the change.
- Modify the structure of the change by adding, removing or reordering changes (Section 7.4.4).
- Add control structures such as iterations and conditionals (Section 7.4.4).

Each type of atomic change has a number of properties that can be edited.Table 7.3 sums up which properties are editable for each kind of atomic change.

Atomic Change	Aim	Possible actions
Creation	Create a new entity of a given type.	Change the kind of entity created. Remove to convert a constant or a variable to a parameter. Create one to do the opposite.
Addition/Removal	Add/Remove a method, variable, class, package from the class, package, system.	Define a condition for the successful addition of the current class, package, method or variable.
Property Change	Change a property (name, superclass, *etc.*) of an entity.	Change the kind of property set. Change the value of the property set (constant or function). Add a success condition.
Insertion/Deletion	Inserts/Remove a statement-level entity in a method body.	Define the insertion/deletion AST pattern for an unlocated variable. Add a success condition. Specify if a selection should be used.

Table 7.3: The properties that can be edited for each atomic change.

Changing properties. By default, a generic property change keeps the property of the original concrete change, but the Change Factory allows it to be either a computation or a demand for user input (for example the user might want to pick a name or a superclass for a given class). To compute properties, the change factory provides the user with a *context* object that can be queried for information during the change's application. Using the context, one can access the values of the parameters and variables during the change execution, as well as the entire state of the program. One can assume all the changes before the current change in the transformation are instantiated and executed. The context has a convenient API to access the most useful queries, such as: current class or method, current method name *etc.* Identifiers can be bound to values in the context and those values can be retrieved later on, to transmit information between changes.

Specifying conditions. The context can be used to define conditions altering the behavior of the change depending on where it is applied (*e.g.,* if the current method does not override another). Conditions can be either preconditions (tested before the change takes place) or post-conditions (tested on the modified entities after the change takes place). If these are not met during instantiation, the change *fails*.

Locating entities with AST patterns. *Unlocated* entities must be found in the ASTs of the methods to which the transformation will be applied. However, dealing with the intricacies of ASTs is one of the overly abstract activities a programmer faces when transforming programs. In addition, building a mental representation is hard since example run-time ASTs are not easily accessible. It is not clear which nodes to look for in the AST, and where they should be located. We address these problems by using the ASTs of the concrete entities on which the recorded changes were applied as an initial *AST pattern* to be incrementally refined by the programmer. An *AST pattern* is an AST enriched with information to relax or constrain its comparison to other ASTs. Furthermore, we minimize the need for the programmer to consider the AST pattern structure by providing a *direct manipulation interface*.

These two features work in concert in the following way: For each insertion or deletion of an unlocated entity, the programmer is presented with the state of the example just before (for a deletion) or after (for an insertion) the change was applied (Figure 7.4, top right panel). The variable inserted or deleted is highlighted to ease focusing. Behind the printed text, the programmer is in fact interacting with a serialized AST pattern of the state of the original example method. When the transformation is applied to another method, the AST of the method will be retrieved and matched against the AST pattern in an attempt to find either a correct place in the AST for an inserted entity, or the ID of an entity in the AST to delete. If the ASTs do not match, the change fails.

If no modification is made to the AST pattern, it will only match the initial method in its initial state. We allow the programmer to relax the constraints in the AST pattern by simply selecting a node or a range of nodes in the text. The tool maintains a mapping from text position to AST nodes, sparing the programmer to manually locate every node. A context menu gives the available constraints for the selected nodes. Upon selection, the constraints are applied to the nodes, and the text in the panel is re-rendered to update the applied constraints. Several constraints can be put on the same node (name, kind, multiplicity, optionality, recursion). The constraints are listed in Table 7.4. Some common sets of constraints are provided as shorthands, such as “allow any method signature” (a method having any name, an unlimited number of arguments and an unlimited number of temporaries), or “any position in the body” (inserts nodes in the pattern matching any other nodes in the relevant position).

Constraint	Effect	Representation
None (default)	Same name and kind (entity type)	foo
Same kind	Same kind as original, any name is possible.	$temporary
Name matches ...	Same kind as original, name matches condition (*e.g.,* the name must start with “set”).	$temporary([n])
Kinds ...	Matches any set of kinds (*e.g.,* any body statement).	$k1|k2 or $*
Optional	May be present.	foo(?)
Forbidden	May not be present. (*e.g.,* specify a counter example).	foo(!)
Unlimited	May be present several times.	foo(+) or foo(*)
No recursion	Ignore any children of the entity (*e.g.,* the presence of an *if* is important, not its contents).	foo(...)
Anything	Any nodes	...
Manual ...	Specified by the programmer with Smalltalk code	foo([])

Table 7.4: Available constraints in AST patterns

Locating variables via selection. Some unlocated variables rely on the user selecting them when the transformation is performed. The change factory allows to specify this as well. When the change is instantiated, a window with the source code of the method to which the variable insertion or deletion is applied opens, asking the user to select the relevant piece of code. One can also ask for the former position of a deleted selection (*e.g.,* to substitute two pieces of code).

Example: property edits. Since it is selection-intensive, "Extract Method with Holes" does not need many property edits. The created variables in "Extract Temporary" (change **A**) need to be named: User input will be requested when the transformation is performed. In change **C**, the newly created method must be named –also via user input– based on the names of the arguments it takes. "Extract Method" must infer the arguments of the arguments: It queries the context to get the set of arguments and temporaries which are referenced both inside and outside of the extracted block of code.

	A: temporary location	**A: assigment location**	**F: temporary location**	**F: assignment location**	**F: reference location**
Original pattern	**exampleMethod:** argument | **<<**value**>>** | value := 42. ^ argument	**exampleMethod:** argument | value | value **<<:=>>** 42. ^ argument	**exampleMethod:** argument | **<<**value**>>** | value := 42. self add: value to: argument. ^ argument	**exampleMethod:** argument | value | value **<<:=>>** 42. self add: value to: argument. ^ argument	**exampleMethod:** argument | value | value := 42. self add: **<<**value**>>** to: argument. ^ argument
Constraints	- any method signature - temporary in last position	- any signature - first body statement	- temporary named as parameter - temporary in any position	- any signature - assignment anywhere in the body - match name in left hand side - the right hand side will be stored	- any signature - position anywhere in the body - name matches parameter - replace declaration with old right hand side
Refined	**$method:** $argument(*****) | $temporary(*) **<<**value**>>** | ...	**$method:** $argument* | $temp* | (...) **<<:=>>** (...) ...	**$method:** $argument* \| **<<**$temporary([n])**>>** | ...	**$method:** $argument* | $temporary* | ... $reference[n] **<<:=>>**	**$method:** $argument* | $temp* \| ... **<<**$reference[n]**>>** ...

Figure 7.5: Initial patterns and resulting constraints

Example: variable Locations. The constant expressions and the extracted code block are user-selected, when the transformation is performed. On the other hand AST patterns need to be defined for the changes dealing with temporaries (A and F). Figure 7.5 shows these patterns.

7.4.4 Composing Changes

The change factory can edit the change structure itself in order to alter its behavior. Possible operations are adding, moving or removing changes and adding control structures to repeat certain changes or apply them conditionally.

Generic Change	Aim	Possible actions
Iteration	Repeat the contained changes on several targets.	Sets the variable that contains the list of targets. Optionally enter an initialization query for this variable.
Conditional	Perform the first contained change that succeeds to apply.	Optionally add a query to choose the change to apply instead of trying them in order.
Optional	Attempts to apply the sub-changes, but does not fails if they do.	None.

Table 7.5: The supported composite generic changes.

Editing the structure. The recorded changes and their generalized counterparts have a tree-like structure. The Change Factory can alter this structure by adding, removing, or reordering the changes. Composite changes can also be merged (two composite changes become one change containing the atomic changes of both), or split (one composite becomes two distinct changes).

These alterations can affect the behavior of the change: Removing a creation changes the role of the unlocated variable or constant which was created to a parameter. Splitting a change allows to decompose a composite change in smaller steps. Some of those might be wrapped in conditional structures afterwards.

Adding Control Structures. We described the control structures in Section 7.2. The Change Factory allows one to select which changes need to be wrapped in a control structure in the tree view of the first panel, and to edit the properties of the change in the second panel (see Table 7.5).

Example. *Structure alteration.* When we recorded the changes to define "Extract Method with Holes", our first selection of the changes contained an extra change, the creation of `exampleMethod:` itself. This change had to be removed to allow `exampleMethod:` to become a parameter to the transformation. Also during recording, changes **C** and **D** were performed as one single change, which had to be split in two.

Iterations. To specify that an unlimited number of holes can be defined, extracted into temporaries and subsequently inlined, we used several iteration constructs: A set of temporary names T is initialized in change **A** using as many selections and input names as the user wants. At the end of the transformation, T is reused in change **F** to inline all the "temporary" temporaries. Change **A** initializes the set of temporaries by asking the user for selections until he or she stops. Also, in changes **C** and **E**, the set of arguments to the method –computed from the context– is used to build the signature of the extracted method and the call to it in the original method.

Calling transformations. In the interest of reuse, the extracted and inlined temporary transformations could be defined separately. "Extract Temporary" takes as parameter a method, and asks the user to select an expression inside it, while "Inline Temporary" takes as parameter a method and a temporary before inlining all references to the variable. Changes **C** and **E** are small enough buildings blocks that they could also be abstracted and reused.

Conditionals. One alternative to using iterations is to define a fixed number of holes, by calling several times the "Extract Temporary" change. At the end of the transformation, the calls to "Inline Temporary" can be wrapped in optional changes, failing it the variable they reference is undefined. It is improbable that many holes will be needed in extracted code, making this "brute force" solution viable.

7.4.5 Testing the Transformation

Due to the exploratory nature of our tool, the boundary between testing and applying a change is fuzzy: An applied change is one which has not been undone. Testing a transformation applies it to the original example, then saves the resulting state, undoes the transformation, applies the initial concrete change, and finally shows the state of the concrete change and the transformation side-by-side. The developer can then compare the current state of the transformation side-by-side with the original example and assess how far he or she is from the desired goal. This allows the developer to quickly evaluate his/her change at the press of a button.

7.4.6 Applying the Transformation

Once defined, transformations can be applied on a case-by-case basis or in a system-wide fashion. The transformation is named and stored in a transformation repository. this repository is made accessible within the IDE through menus for case-by-case usage. When browsing the system, the user can decide to apply a transformation in the repository to the entities he is currently viewing. This is the preferred way to use a transformation such as "Extract Method with Holes".

Larger transformations can be applied system-wide, on larger set of entities, using a Change Applicator tool with which one can select a larger set of entities in the system and display the result. For instance, the Change Applicator makes it easy to select all the classes in the packages, or all the methods in a class.

7.5 Additional Examples

Example-based program transformations are useful in a variety of contexts, beyond“Extract Method With Holes” illustrated above. We describe two additional examples.

7.5.1 Defining informal aspects

If an application is designed from the ground up to incorporate aspects, the design allows it clearly. However, when introducing an aspect in an existing system, it is not always clear where to introduce the aspect. The developers have to localize code related to the concern they want to separate, isolate it in an aspect, and determine general rules to define a pointcut before weaving the aspect back in the system.

In contrast, our approach allows one to define an aspect informally, and to initially collect its join points manually. The transformation definition represents the advice of such an informal aspect, while the entities to which the transformation is applied are its join points. Applying the transformation corresponds to weaving the aspect, and undoing it removes the concern from the system. Since the transformation is reified and its effects upon instantiation are reified and documented, unweaving an aspect is easy.

If an aspect is worth keeping, the extensive listing of the join points could be used as a basis for a more formal definition of a pointcut. Techniques employed in aspect mining could be used to assist such a process [BvDT05; BCH$^+$05].

Defining aspects with the Change Factory bears resemblance to Fluid AOP, a recent variant of AOP where aspects are not implemented by an aspect oriented programming language, but by tools integrated with the IDE [HK06].

Informal aspects exemplified. An example is the running example of Section 7.3, the simple logging aspect. To define such an aspect involves the following steps:

1. Record the insertion of a logging statement at the start of a method, calling the logging facility with a literal specifying the name of the method.
2. The system deduces that the only parameter of the example is the method to which the transformation should be applied.
3. The programmer edits the position of the logging statement to match the start of any method. The “name” property of the string that is printed is computed by a context query which includes the name of the method to which the transformation is applied.
4. The programmer might actually record several examples, depending on the number of arguments. If the method has arguments, the values of these could be logged. These changes could then be wrapped in a conditional change to choose the correct transformation to apply to a system.
5. The change can then be applied to all the join points in the system.

7.5.2 Clone Management

Several researchers have claimed that code clones are sometimes best “left alone” [KSNM05; KG06]. They point that in certain conditions, refactoring a set of clones to remove the duplication is either not possible due to language limitations or too expensive. It can even be harmful if it leads to over-abstraction, or if the clones are destined to grow apart eventually. To assist the programmer when dealing with duplicated, unrefactorable code, linked editing [TBG04] and clone tracking tools [DER07] have been proposed. These tools maintain a list of clones, and when one of these is edited, propose to the user to change the other clones in the same way.

When propagating changes from one clone to the others, clone tracking tools usually work at the text level. Since we record changes at the AST level, we can propagate changes to clones at a syntactic level rather than a textual one. Moreover, our AST pattern definitions are more tolerant with differences between clones (inserted/deleted statements, renamings, *etc.*), since these can be relaxed when interacting with the AST patterns. As such, we believe our tool could manage situations better than current linked editing tools.

One advantage that linked editing tools have over our current implementation is a larger degree of automation. Currently, our system lacks the ability to detect clones, so the developer has to propagates changes manually. However integrating a clone-detection tool is possible. The specification of AST pattern constraints is more work than other tools, but is more flexible when broadcasting the changes. We still have to investigate whether and to which extent having several examples (*i.e.,* all the clones in a clone group), allows one to automatically relax constraints in the AST patterns to fit as many members of the group as possible.

Linked Editing Exemplified. Our own tools are not exempt from cloning. For instance, a design decision forced us to duplicate some code during the implementation of the Change Factory. Our change metamodel features concrete and generic changes. Since Smalltalk does not provide multiple inheritance, the implementation of generic atomic changes leads to problems.

The two possible solutions both involve code duplication. They are shown in Figure 7.6. We chose extension A. During implementation, we had to change how concrete changes were instantiated. Previously, a concrete change was simply returned by the generic atomic change. We found best to directly execute the concrete change during instantiation, before returning it. This spared the composite changes to do it themselves but forced us to update all 6 versions of the `instantiateIn:` method [4].

As seen in Figure 7.7, each method creates a new change, sets it up, and returns it. Setting it up differs for each change. In addition, the change has a different name in each method for clarity, and due to inconsistencies, the name of the argument differs in some of the methods. When recording an instance of the change, the system will deduce that the contents of the return statement was deleted, and another statement was inserted. It will also identify that

[4]If we had a common superclass for all the changes (extension B), we could have put this behavior in it.

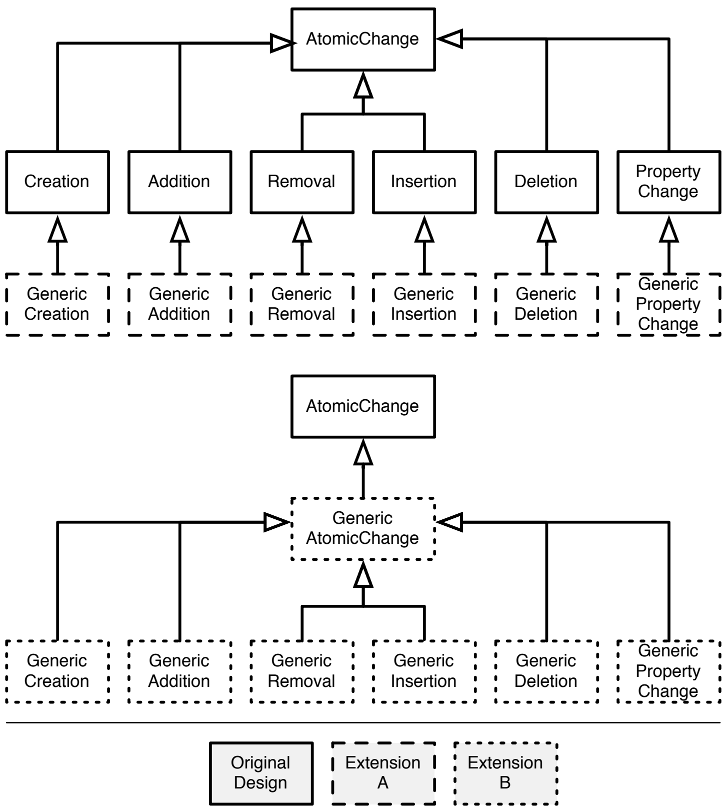


Figure 7.6: Two possible generic change designs

there is one parameter to the change, the method to which it is applied. To define an AST pattern matching the needed editions, the user has to specify the following constraints:

- The argument's name varies, and must be stored (as X).
- The temporary's name varies, and must be stored (as Y).
- Any number of body statements can follow.
- When a return statement with the temporary is encountered, the temporary must be deleted.
- When an empty return statement is encountered, a message send must be inserted, with receiver X and argument Y.

```
class: GenericCreation

instantiateIn: aSWModelView
      | creation |
      self entity id: ID gen.
      creation := SWCreation new.
      creation entity: self entity id.
      creation kind: self kind.
      ^ aSWModelView execute: creation
      ~~^ creation~~
```

```
class: GenericAddition

instantiateIn: aSWModelView
      | addition |
      addition := SWAddition new.
      addition entity: self entity id.
      addition parent: self parent id.
      ^ aSWModelView execute: addition
      ~~^ addition~~
```

```
class: GenericInsertion

instantiateIn: aView
      | insertion state |
      insertion := SWInsertion new.
      insertion entity: self entity id.
      insertion root: self root id.
      self useSelection
            ifTrue: [self initializeFromSelection: insertion in: aView]
            ifFalse: [(entity isConstantIn: self composite) ifFalse: [
                        state := aView stateOf: root id.
                        self updatePatternIn: aView.
                        pattern matches: state.]].
      insertion under: self under id.
      insertion after: self after id.
      ^ aView execute: insertion
      ~~^ insertion~~
```

```
class: GenericPropertyChange

instantiateIn: aSWModelView
      | propertyChange |
      propertyChange := SWPropertyChange new.
      propertyChange entity: self entity id.
      propertyChange property: self property.
      propertyChange value:
            (self valueIn: aSWModelView).
      ^ aSWModelView execute: propertyChange
      ~~^ propertyChange~~
```

Figure 7.7: Sample clones in the Change Factory

7.6 Towards Transformation Integration and Evolution

In this section, we describe how transformations are integrated in our model, and what consequences this has for the system in terms of comprehension and evolution.

7.6.1 Transformation Integration

Transformation integration is enabled by a very simple addition to the model. Since a transformation is a change generator, it has control on how it generates them, and is free to add metadata to the change for this specific instantiation.

Whenever a transformation is defined, it is stored in a transformation repository for future usage. Each change generated by the transformation is also tagged with an identifier linking it to the transformation that generated it. These changes are stored in the change repository like any other change, where they are explicitly linked to the transformation that produced it, and the values of the parameters that were given to the transformation.

This is enough to fully integrate the transformations in the evolution, as the changes they generate can be treated either as normal changes when replaying them, or generated changes when a deeper processing is needed.

7.6.2 Transformation Comprehension

Each transformation application is stored and explicitly documented in the change repository. This allows transformations to be treated in the same way as refactorings for program comprehension purposes. We used refactorings for program comprehension in Chapter 5. They gave context to surrounding changes, and were reviewed more quickly since they were automated and marked as such.

The same treatment is possible for program transformations. In particular, the concrete changes in the transformation can be easily traced back to their abstracted purpose, the transformation application. Understanding a systematic change to each method of a given class is immediate when it can be traced back to the application of the “Informal Logging” aspect. All the changes related to the aspect are clearly delimited.

Transformation application can be treated differently than other changes to compute metrics. Change-based metrics can be defined to better understand sessions, listing transformations separately. Logical coupling measurements may want to weight the changes caused by transformations differently.

7.6.3 Transformation Evolution

In the face of changing requirements, some transformations might no longer be needed, need to be changed, or applied to new parts of the system. Dig and Johnson reported that 80% of changes breaking the APIs of framework are due to applications of automated transformations, namely refactorings [DJ05]. This shows that even behavior-preserving transformations such as refactorings cause problems in the evolution of systems.

Documenting the application of transformations is a first step towards a better support of their evolution. The first immediate usage is that if a transformation needs to be changed, the places that are affected in the system are immediately known, as the changes it generated can be searched in the history. Each of these can then be reviewed to determine how and if it needs to change in the face of the new requirements. A newer transformation can subsequently be defined and applied to these new locations.

A transformation can be undone in all the places where it was applied. If some of those were changed afterwards, undoing the transformation might not be possible without undoing these changes. The list of conflicting places can be brought up, and manual inspection can determine how to deal with them. How much of that process can be automated is a question for which we do not have an answer yet.

Finally, documenting where the transformation was applied means that one also knows where it was *not* applied. This makes it easier to apply it to parts of the system which need the transformation due to updated requirements. A use case for this is the case of refactorings in frameworks: A refactoring may have renamed all the references to an entity inside the framework, but client code needs to be updated as well.

7.7 Discussion

7.7.1 Change-based Program Transformation

Impact of Composition

We allow *sequences* of changes to be recorded and specified. This eases the definition of transformations affecting several entities. Several transformation tools, such as iXj [BGH07] or the Rewrite Tool [RB04], are based on pattern-matching to provide a concrete syntax to ease transformation definition. They however operate on a single pattern at once, which limits the extent of the transformations they can define.

Applicability

Our prototype is implemented in Smalltalk, a language with a simple and consistent syntax. Applying it to a more complex language like Java, which is typed and includes generic types, might pose some issues. We think it is feasible, since many transformation tools exist for Java: The existence of such tools might help us porting our approach. However, whether the approach is as usable in such a context is yet to be determined. In our approach, the type of an entity is modeled as a property, which may need additional constraints to enforce the type system.

Expressivity

We did not directly evaluate the expressivity of our approach, *i.e.,* if every kind of transformation can be expressed with it. The change-based program transformation and their applications could span an entire thesis topic in itself. In this chapter, we focused on the definition of transformation and their integration in the overall Change-based Software Evolution approach. We undertook a feasibility study in which we applied it to three examples. We consider the expressivity more of a pure program transformation problem, whereas our primary objective was to extend Change-based Software Evolution with program transformation support in a natural and integrated fashion.

We are however confident that our approach is expressive enough. One of the examples we selected was a more complex variant of the “Extract Method” refactoring, which Fowler describes as the *Refactoring Rubicon, i.e.,* a refactoring that has the necessary complexity to indicate that the approach supports other kinds of refactorings.

Moreover, our approach shares a lot with other transformation tools. It merely describes the transformations in a more concrete way. Other problems like expressing conditions can be done in the same way other tools do if needed. The fact that other tools are expressive enough to handle more kinds of transformation is an indication our approach is expressive as well.

Behavior preservation

Unlike refactorings, behavior preservation is not guaranteed. Our tool will require programmer supervision to ensure the results are correct. Some transformations could be recognized as behavior-preserving once the needed analyses are defined.

7.7.2 Example-based Program Transformation

Example-based

Our approach uses two kinds of entities: Domain entities comprising the AST of the system, and changes applied to them. Using our change representation on top of the AST allows us to infer from a single example which entities are parameters to the change, and which ones are unlocated. Furthermore, we display ASTs to allow the programmer to work directly with the concrete syntax of the system, instead of having to learn a dedicated syntax to match entities. Since those ASTs are also extracted from examples, one does not start over, but abstracts away from the concrete example at hand.

Quality of the examples

When coding, programmers often make errors and backtrack. These digressions are recorded in our changes and are unnecessary when generalizing the change. To address this one can use our change-editing facilities to remove undesired changes from the sequence. Alternatively, the example can be "replayed", *i.e.,* re-recorded to avoid the quirks introduced the first time around.

Related Transformation Approaches

Several approaches have used concrete syntax to ease the definition of transformations. Stratego/XT is a program transformation tool, which has seen applications in defining language extensions. Visser argues that manipulating ASTs of programs is too complex for many applications, and proposes a scheme to instead use the concrete syntax of the programming language [Vis02]. The result is a transformation language with both Stratego and the concrete syntax.

In the same vein, De Roover *et al.* [RDB+07] introduced a concrete layer on top of a logic programming language, similar to Java source code, with variables to be matched prefixed by question marks. The rationale is to simplify matching structures by hiding the AST where possible.

Due to its extensive IDE integration, the language-based approach closest to ours is iXj by Boshernitsan *et al.* [BGH07], an interactive IDE extension to transform Java programs supported by a visual programming language. ASTs are represented graphically. The transformation still has to be written with only the starting state specified (via selection).

Roberts and Brant describe the Rewrite Tool [RB04] which uses pattern matching to implement arbitrary transformations. However the patterns defined in the transformations can only refer to one entity at a time and must be written from scratch in a dedicated language.

Program Transformation in Model-Driven Engineering

Transformation is a prominent concept in model-driven engineering. Several model transformation languages have been defined, such as MTL, Xion and Kermeta [MFV+05], or ATL [JK05]. However, they have a different target than our approach, since they transform abstract models of programs, and not the programs themselves.

Another tendency found in model transformation is to use example for the definition of transformations. This approach was pioneered by Varró [Var06]. His approach requires an example of a source model and a target model and infers the changes in them. This approach has also been adopted by Wimmer *et al.* [WSKK07], and Kessentini *et al.* [KSB08].

7.7.3 Integrating Transformations in The Evolution

Approaches Integrating Refactorings in the Evolution

Several approaches address the problem of refactoring frameworks. When a framework is refactored, its users may experience API-breaking changes [DJ05]. Several approaches either record the refactoring application and apply it on client code (work by Henkel and Diwan [HD05], Ekman and Asklund [EA04], Dig *et al.* [DMJN07]), or recover the refactorings from SCM archives (work by Weißgerber and Diehl [WD06], Dig *et al.* [DCMJ06]). Other generate code adaptors at the source or binary level, such as the Comeback approach by Savga *et al.* [SRG08] and the ReBa approach by Dig *et al.* [DNMJ08].

A limitation of these approaches is that they only consider refactorings, whereas our approach has the potential to be applied to every kind of program transformation, whether they are behavior-preserving or not. In addition, our approach allows full integration of the transformations in the evolution of the system. The only other approach that promotes integration in the history are SCM system reifying refactorings [EA04], [DMJN07]. They are however limited to only refactorings, and do not describe other changes to the system, storing merely versions. Their integration is hence far from complete.

7.8 Summary

By automating repetitive changes, program transformation is one of the most useful tool to support software engineering. In this chapter, we investigated if Change-based Software Evolution could be extended with program transformations. We divided this problems in three sub-problems, and found that:

- Change-based Software Evolution naturally supports automated program transformations. Each atomic change is in essence a constant program transformation. We successfully extended our model to include high-level, parametrized program transformations. When applied to a set of parameters on a system, these generate a set of concrete transformations corresponding to the actual change to perform. We simply added a layer on top of our model which does not affect the bottom layer.
- Change-based Software Evolution eases the definition of program transformations. One of the key factors when dealing with abstract concepts such as program transformations is the need of concrete examples. We found that recording a concrete change as an example of a transformation and subsequently generalizing it was a natural process. The structure of the recorded change acts as a checklist of what needs to be generalized. Each individual change can be subsequently customized. The concrete examples can also be generalized to define where the transformation should be applied.
- Transformations can be integrated in the evolution. In the same way our approach recognizes refactorings and uses them to assist program comprehension, applications of transformations –being generated changes– can be traced and used for program understanding later on. In addition, tagging changes as parts of transformations enables transformation evolution: If a transformation needs to be updated, instances of its application can be recalled and reviewed.

An aspect which we explored only partially is automation: Our transformation definition approach is at the moment semi-automated. We need to investigate how much further it can be automated, through the use of more than one example. Possible enhancements are to use multiple examples to automate the definition of the AST patterns and conditions, automating code clone management, and automating the evolution of transformations by changing the locations were they were previously applied.

Chapter 8

Evaluating Recommendations for Code Completion

Recommender systems assist programmers but must be evaluated with care: An inaccurate recommender system will be harmful to a programmer's productivity. An example is code completion. Code completion is a productivity tool used by every programmer. Code completion is seldom improved because it has reached a local maximum with the information available in current IDES. Furthermore, the accuracy of a completion engine is hard to assess, besides manual testing.

We use data provided by Change-based Software Evolution to both define a benchmark to comprehensively evaluate the quality of a code completion engine and to improve completion engines. By using program history as an additional information source, we significantly increase the accuracy of a completion engine. The probability that the match a programmer is looking for shows up in the completion tool in a top spot can reach 80%, even with very short completion prefixes.

8.1 Introduction

This chapter is the first (along with Chapter 9) where we assess the usefulness of Change-based Software Evolution to implement and evaluate recommender systems. Recommender systems aim to improve the productivity of programmers when building and maintaining systems, by assisting them while they change them. Such assistance can take several forms: Recommending which entities to change next (the focus of Chapter 9), filtering out potentially useless entities, or pointing out shortcomings in the system's design, such as duplication or code smells. In all cases, recommender systems have to be accurate: Incorrect predictions will actually slow down the programmer, forcing him to invest significant time and cognition into false leads.

Recommender systems must hence be evaluated with care. Such an evaluation is however not simple. The most natural evaluation strategy is the human subject study, where test subjects are monitored with or without using the tool while performing a given task. While giving a relatively high confidence in the results, these evaluations are very long and expensive to set up.

An alternative evaluation strategy is the benchmark. Eliott-Sim *et al.* have shown that benchmarks have the potential to dynamize a research community by making evaluations easier to perform, compare and replicate [SEH03].

A benchmark is hence desirable at least as a pre evaluation technique before a human subject study begins. However, constituting the data corpus the benchmark uses may be difficult, depending on the data needed by the tools.

A recommender system lacking such a benchmark is code completion. In 2006, Murphy *et al.* published an empirical study on how 41 Java developers used the Eclipse IDE [MKF06]. One of their findings was that each developer in the study used the code completion feature. Among the top commands executed across all 41 developers, code completion came sixth with 6.7% of the number of executed commands, sharing the top spots with basic editing commands such as copy, paste, save and delete. It is hardly surprising that this was not discussed much: Code completion is one of those features that once used becomes second nature. Nowadays, every major IDE features a language-specific code completion system, while any text editor has to offer at least some kind of word completion to be deemed usable for programming.

Despite the wide usage of code completion by developers, research in improving code completion has been rare, due to the difficulty of evaluating it with respect to the expected improvement.

In this chapter, we test the use of Change-based Software Evolution to define benchmarks for recommender systems where data was not previously available, through the example of code completion. In essence, our benchmark replays the entire development history of the program and calls the completion engine at every step, comparing the suggestions of the completion engine with the changes that were actually performed on the program. We also investigate if and how Change-based Software Evolution can be used to improve the recommender system themselves. Initial evidence leads us to believe so: We saw in Chapter 4 and Chapter 5 that Change-based Software Evolution highlights the relationships between program entities, and in Chapter 6 that this translates in measurable improvements. Using the accurate data Change-based Software Evolution provides may also demonstrably improve recommender systems. With our benchmark as a basis for comparison, we define alternative completion algorithms which use change-based historical information to different extents, and compare them to the default algorithm which sorts matches in alphabetical order.

Contributions. The contributions of this chapter are:

- The definition of a benchmark for code completion engines, based on replaying fine-grained program change histories and testing the completion engine at every opportunity.

- The definition of several code completion algorithms, and their evaluation with the benchmark we defined.

Structure of the chapter. Section 8.2 compares human subject studies with benchmarks with respect to ease of creating and sharing experiments. Section 8.3 details existing code completion algorithms and their relative lack of evaluations. Next, Section 8.4 presents the benchmarking framework we defined to measure the accuracy of completion engines. In Section 8.5 we evaluate several code completion algorithms, including algorithms using recent change information to fine-tune their recommendations. Finally, after a discussion in Section 8.6, we conclude in Section 8.7.

8.2 The Cost of Human Subject Studies

Human subject studies have a long tradition as an evaluation method in software engineering for methodologies and tools. They usually involve two groups of people assigned to perform a given task, one using the methodology under study, and a control group not using it. The performance of the groups are then measured according to the protocol defined in the study, and compared with each other. Hopefully, the methodology under study provides an improvement in the task at hand. To have confidence in the measure, a larger sample of individual is needed to confirm a smaller increase. If they provide usually high confidence in their results, human subject studies have drawbacks:

- They are very time-consuming and potentially expensive to set up. Dry runs must be performed first, so that the experiment's protocol is carefully defined. Volunteers have to be be found, which may also require a monetary compensation. The most extreme case in recent history is the pair programming study of Arisholm *et al.*, which tested –and compensated– 295 professional programmers [AGDS07].

- Since they are expensive and time-consuming to set-up, they are as difficult to reproduce. The original authors need to document their experimental set-up very carefully in order for the experiment to be reproduced. Lung *et al.* documented [LAEW08] the difficulties they encountered while reproducing a human subject study [DB06].

- The same time-consuming issues make them unsuited for incremental refinement of an approach, as they are too expensive to be run repeatedly. In addition, a modest increment on an existing approach is harder to measure and must be validated on a higher sample size, making the study even more expensive.

- Comparing two approaches is difficult, as it involves running a new experiment pitting the two approaches side by side. The alternative is to use a common baseline, but variations in the set-up of the experiment may skew the results.

- In the case of tools, they include a wide range of issues possibly unrelated to the approach the tool implements. Simple UI and usability issues may overshadow the improvements the new approach brings.

Another evaluation methodology is the benchmark. A benchmark is a procedure designed to (most of the time) automatically evaluate the performance of an approach on a dataset. A benchmark hence carefully delimit the problem to be solved in order to reliably measure performance against a well-known baseline. The outcome of a benchmark is typically an array of measurement summing up the overall efficiency of the approach. An example is the CppETS benchmark, for C++ fact extractors in the context of reverse engineering [SHE02]. Its data corpus is made of several C++ programs exercising the various capabilities of fact extractors. A fact extractor can be run on the data set, and will return the list of facts it extracted, which can be compared with known results. The fact extractor is then reliably evaluated. A benchmark has the following advantages over a human subject study:

- Automated benchmarks can be run at the press of a button. This allows each experiment to be ran easily, and reran if needed. This considerably eases the replication of other people's experiments.

- Benchmarks usually score the approaches they evaluate, making it trivial to compare an approach to another.

- The simplicity of running an experiment and the ease of comparison makes it easy to measure incremental improvements.

- Benchmarks test a restricted functionality, and if automated are impervious to usability issues.

- Making them more extensive is as simple as adding data to their current data corpus.

In a nutshell, the strength of the benchmark are the weaknesses of the human subject study. As Sim *et al.* explained, these advantages dynamize the activity of a research community that uses a benchmark [SEH03].

However, creating the benchmark itself and the data corpus it uses represents a considerable amount of work. For the C++ fact extractor benchmark, it presumably involved a manual review of the C++ programs in the dataset to list the expected facts to be extracted. In the case of other systems, the tasks may be too large to be worthwhile.

8.3 Current Approaches to Code Completion

In the following, we focus on the completion *engine*, *i.e.,* the part of the code completion tool which takes as input a token to be completed and a context used to access all the information necessary in the system, and outputs an ordered sequence of possible completions. We describe code completion in three IDEs: Eclipse (for Java), Squeak, and VisualWorks (for Smalltalk).

8.3.1 Code Completion in Eclipse

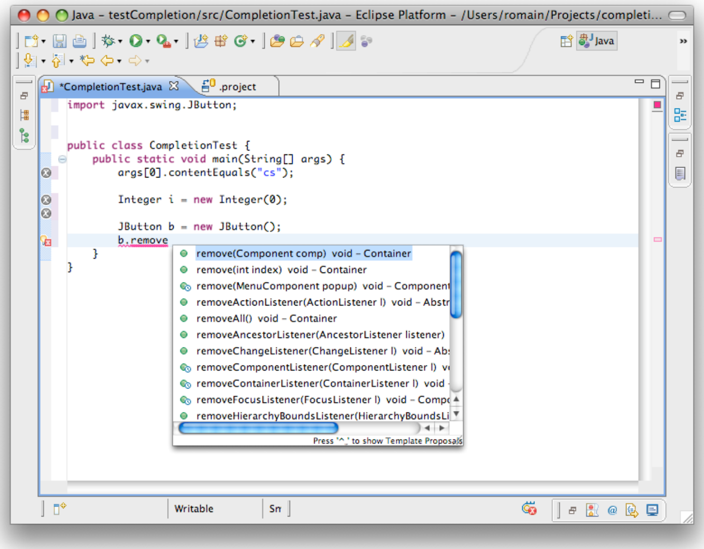
Code completion in Eclipse for Java is structure-sensitive, *i.e.,* it can detect when it completes a variable or a method name. It is also type-sensitive: If a variable is an instance of class String, the matches returned when completing a method name will be looked for in the classes `String` and `Object`, *i.e.,* the class itself and all of its superclasses.

Figure 8.1 shows Eclipse's code completion in action: The programmer typed "remove" and attempts to complete it. The system determines that the object to which the message is sent is an instance of `javax.swing.JButton`. This class features a large API of more than 400 methods, of which 22 start with "remove". These 22 potential matches are all returned and displayed in a popup window showing around 10 of them, the rest needing scrolling to be accessed. The matches are sorted in alphabetical order, with the shorter ones given priority (the first 3 matches would barely save typing as they would only insert parentheses).

This example shows that sometimes the completion system, even in a typed programming language, can break down and be more a hindrance than an actual help. As APIs grow larger, completion becomes less useful, especially since some prefixes tend to be shared by more methods than other: For instance, more than a hundred methods in `JButton`'s interface start with the prefix "get".

8.3.2 Code Completion in VisualWorks

VisualWorks is a Smalltalk IDE sold by Cincom. Since Smalltalk is a dynamically typed language, VisualWorks faces more challenges than Eclipse to propose accurate matches. The IDE can not make any assumption on the type of an object since it is determined at runtime only, and thus returns potential candidates from all the classes defined in the system.


Java – testCompletion/src/CompletionTest.java – Eclipse Platform – /Users/romain/Projects/completi…
Java
*CompletionTest.java
.project
Writable
Sn

```
import javax.swing.JButton;


public class CompletionTest {
    public static void main(String[] args) {
        args[0].contentEquals("cs");

        Integer i = new Integer(0);

        JButton b = new JButton();
        b.remove
    }
}
```

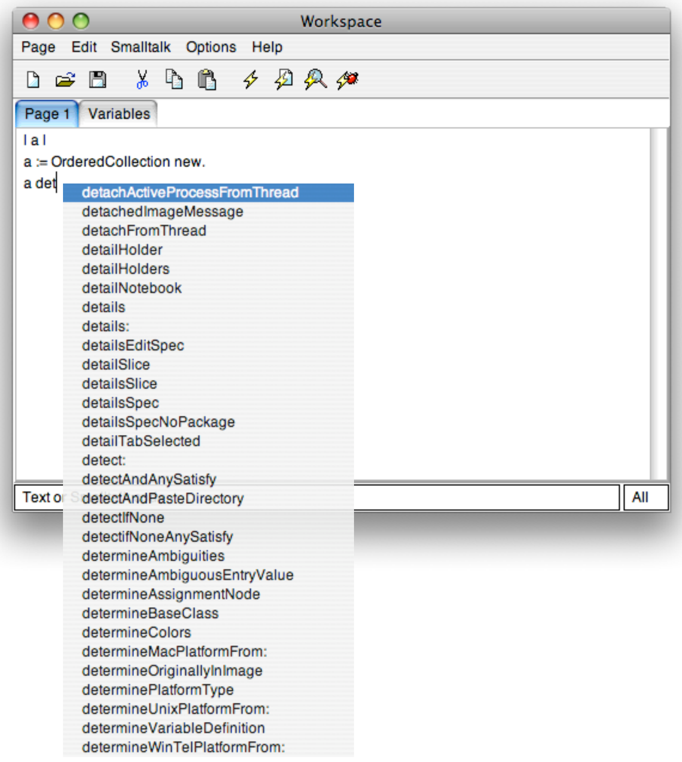
```
remove(Component comp)  void – Container
remove(int index)  void – Container
remove(MenuComponent popup)  void – Component
removeActionListener(ActionListener l)  void – Abstr
removeAll()  void – Container
removeAncestorListener(AncestorListener listener)
removeChangeListener(ChangeListener l)  void – Ab
removeComponentListener(ComponentListener l)  v
removeContainerListener(ContainerListener l)  void
removeFocusListener(FocusListener l)  void – Comp
removeHierarchyBoundsListener(HierarchyBoundsLi
Press '^,' to show Template Proposals
```

Figure 8.1: Code completion in Eclipse

Since Smalltalk contains large libraries and is implemented in itself, the IDE contains more than 2600 classes already defined and accessible initially. These 2600 classes total more than 50,000 methods, defining around 27,000 unique method names, *i.e.,* 27,000 potential matches for each completion. The potential matches are presented in a menu, which is routinely more than 50 entries long (see Figure 8.2). As in Eclipse, the matches are sorted alphabetically, but the sheer number of possible matches renders the system very hard to use.

8.3.3 Code Completion in Squeak

Squeak's completion system has two modes. The normal mode of operation is similar to VisualWorks: Since the type of the receiver is not known, the set of candidates is searched for in the entire system. However, Squeak features an integration of the completion engine with a type inference system, Roel Wuyts' RoelTyper [Wuy07]. When the type inference engine finds a possible type for the receiver, the completion is equivalent to the one found in Eclipse. Otherwise matches are searched in the entire system (3000 classes, 57,000 methods totaling 33,000 unique method names). In both cases matches are alphabetized.



```
| a |
a := OrderedCollection new.
a det
```

Figure 8.2: Code completion in VisualWorks

8.3.4 Code Completion in Eclipse with Mylyn

An alternative completion engine for Eclipse is shipped with the Mylyn tool. It leverages Mylyn's degree-of-interest model to prioritize entities with a high degree-of-interest value in Eclipse's completion menu. However, it was only mentioned in passing as an add-on to the Mylyn tool, and never fully evaluated [KM06]. Its effect in the overall productivity enhancements provided by Mylyn's DOI could not be measured.

8.3.5 Optimistic and Pessimistic Code Completion

All these algorithms, except Mylyn, have the same shortcoming: the match actually looked for may be buried under a large number of irrelevant suggestions because the matches are sorted alphabetically. The only way to narrow it down is to type a longer completion prefix which diminishes the value of code completion. To qualify completion algorithms, we reuse the "pessimistic/optimistic" analogy first employed in Software Configuration Management. Versioning systems have two major ways to resolve conflicts for concurrent development [CW98]. *Pessimistic version control* prevents any conflict by forcing developers to lock a resource before using it. In *optimistic version control*, conflicts are possible but several developers can freely work on the same resource. The optimistic view states that conflicts do not happen often enough to be counter-productive. Today, every major versioning system uses an optimistic strategy [ELvdH$^+$05].

We characterize current completion algorithms as "pessimistic": They expect to return a large number of matches, and order them alphabetically. The alphabetical order is the fastest way to look up an individual entry among a large set. This makes the entry lookup a non-trivial operation: As anyone who has ever used a dictionary knows, search is still involved and the cognitive load associated with it might incur a context switch from the coding task at hand.

In contrast, an "optimistic" completion algorithm would be free of the obligation to sort matches alphabetically, under the following assumptions:

1. The number of matches returned with each completion attempt are limited. The list of matches must be very quick to be checked. Our implementation limits the number of matches returned to 3.

2. The match the programmer is looking for has a high probability of being among the matches returned by the completion engine. Even if checking a short list of matches is fast, it is pointless if the match looked for is not in it.

3. The completion prefix needed to have the correct match with a high probability should be short to minimize typing. With a 10 character prefix, it is an easy task to return only 3 matches and have the right one among them.

To sum up, an optimistic code completion strategy seeks to maximize the probability that the desired entry is among the ones proposed, while minimizing the number of entries returned, so that checking the list is fast enough. It attempts to do so even for short completion prefixes to minimize the typing involved by the programmer. The question then is to find out whether optimism is a sound completion strategy: How can we be sure that a given algorithm has a high enough probability of giving the right answer?

8.4 A Benchmark For Code Completion

To describe our benchmark for code completion, we reuse the format we used in our prediction benchmark for logical coupling measurements in Chapter 6. We first motivate the need for the benchmark, then describe how it is run, the way the results are evaluated and presented, before presenting the corpus that constitutes the dataset used by the benchmark.

8.4.1 Motivation

In our review of current approaches to code completion, we noticed that all but one approaches were very similar. Improvements to completion are rare: The only one we found was mentioned as a side remark in a more general article, and was not evaluated by itself.

This does not mean that code completion cannot be improved, far from it: The set of possible candidates (referred from now on as suggestions or matches) returned by a code completion engine is often inconveniently large. The match a developer is actually looking can be buried under many irrelevant suggestions. If spotting it takes too long, the context switch risks breaking the flow the developer is in. Given the limitations of current code completion, we argue that there are several reasons for the lack of work done to improve it:

1. *Local Maximum.* There is no obvious way to improve language-dependent code completion: Code completion algorithms already take into account the structure of the program, and if possible the APIs the program uses. To improve the state of the art, additional sources of information are needed.

2. *Hard to Measure.* Beyond obvious improvements such as using the program structure, there is no way to assert that a completion mechanism is “better” than another. A standard measure of how a completion algorithm performs compared to another on some empirical data is missing, since the data itself is not there. The only possible assessment of a completion engine is to manually test selected test cases.

3. *If it ain’t broke, don’t fix it.* Users are accustomed to the way code completion works and are resistant to change. This healthy skepticism implies that only a significant improvement over the default code completion system can change the status quo.

Ultimately, these reasons are tied to a single one: Code completion is “as good as it gets” with the information provided by current IDEs. To improve it, we need additional sources of information, and provide evidence that the improvement is worthwhile. A human subject study of a code completion tool seems disproportionate.

However, code completion is a prime candidate for a benchmark-based evaluation, since the problem can be easily reduced to the ranking of matches returned by the completion engine. What is missing is realistic data of completion usage.

Change-based Software Evolution provides it. The idea behind our benchmark is to replay the change history of programs while calling the completion engine as often as possible. Since the information we record in our repository is accurate, we can simulate a programmer typing the text of the program while maintaining its structure as an AST. While replaying the evolution of the program, we can potentially call the completion engine at every keystroke, and gather the results it would have returned, as if it had been called at that point in time. Since we represent the program as an evolving AST, we are able to reconstruct the context necessary for the completion engine to work correctly, including the structure of the source code. For instance, the completion engine is able to locate in which class it is called, and therefore works as if under normal conditions.

8.4.2 Procedure

The rationale behind the benchmarking framework is to reproduce as closely as possible the conditions encountered by the completion engine during its actual use. To recreate the context needed by the completion engine at each step, we execute each change in the change history to recreate the AST of the program. In addition, the completion engine can use the actual change data to improve its future predictions. To measure the completion engine's accuracy, we use algorithm 3.

```
Input: Change history, completion engine to test
Output: Benchmark results
$results$ = newCollection();
foreach Change ch in Change history do
    if methodCallInsertion(ch) then
        $name$ = changeName($ch$);
        foreach $prefix$ of name between 2 and 8 do
            $entries$ = queryEngine($engine$, $prefix$);
            $index$ = indexOf($entries$, $name$);
            add($results$, length($prefix$), $index$);
        end
    end
    processChange($engine$,$ch$);
end
```

Algorithm 3: The benchmark's main algorithm

While replaying the history of the system, we call the completion engine whenever we encounter the insertion of a statement including a method call. To test its accuracy with variable prefix length, we call the engine with every prefix of the method name between 2 and 8 letters –a prefix longer than this would not be worthwhile. For each prefix, we collect the list of suggestions, look up the index of the method that was actually inserted in the list, and store it in the benchmark results. One can picture our benchmark as emulating the behavior of a programmer compulsively pressing the completion key. The benchmark does not ask for predictions for changes done as part of refactorings or other code transformations, as these were not initially performed by a developer.

Using a concrete example, if the programmer inserted a method call to a method named `hasEnoughRooms()`, we would query the completion engine first with "ha", then "has", then "hasE", ..., up to "hasEnoughR". For each completion attempt we measure the index of `hasEnoughRooms()` in the list of results. In our example, `hasEnoughRooms()` could be 23rd for "ha", 15th for "has" and 8th for "hasE".

It is possible that the correct match is not present in the list of entries returned by the engine. This can happen in the following cases:

1. The method called does not exist yet. There is no way to predict an entity which is not known to the system.

2. The match is below the cut-off rate we set. If a match is at an index greater than 10, we consider that the completion has failed as it is unlikely a human will scroll down the list of matches. In the example above, we would store a result only when the size of the prefix is 4 (8th position).

In the latter case we record that the algorithm failed to produce a useful result. When all the history is processed, the results are stored, before being evaluated.

8.4.3 Evaluation

To compare algorithm with another, we need a numerical estimation of its accuracy. Precision and recall are often used to evaluate prediction algorithms. For completion algorithms however, the ranking of the matches plays a very important role. For this reason we devised a grading scheme giving more weight to both shorter prefixes and higher ranks in the returned list of matches. For each prefix length we compute a grade G_i, where i is the prefix length, in the following way:

$$G_i = \frac{\sum_{j=1}^{10} \frac{results(i,j)}{j}}{attempts(i)} \tag{8.1}$$

Where $results(i, j)$ represents the number of correct matches at index j for prefix length i, and $attempts(i)$ the number of time the benchmark was run for prefix length i. Hence the grade improves when the indices of the correct match improves. A hypothetical algorithm having an accuracy of 100% for a given prefix length would have a grade of 1 for that prefix length.

Based on this grade we compute the total score of the completion algorithm, using the following formula which gives greater weight to shorter prefixes:

$$S = \frac{\sum_{i=1}^{7} \frac{G_{i+1}}{i}}{\sum_{k=1}^{7} \frac{1}{k}} \times 100 \tag{8.2}$$

The numerator is the sum of the actual grades for prefixes 2 to 8, with weights, while the denominator in the formula corresponds to a perfect score (1) for each prefix. Thus a hypothetical algorithm always placing the correct match in the first position, for any prefix length, would get a score of 1. The score is then multiplied by 100 to ease reading.

8.4.4 Result Format

First, we mention the overall accuracy score of the algorithm (out of 100). We also display the data in a more detailed format to facilitate analysis. We provide a table showing the algorithm's results for prefixes from 2 to 8 characters. Each column represents a prefix size. The results are expressed in percentages of accurate predictions for each index. The first row gives the percentage of correct prediction in the first place, ditto for the second and third. The fourth row aggregates the results for indices between 4 and 10. Anything more than 10 is considered a failure since it would require scrolling to be selected. Failures are indicated in the bottom row.

8.4.5 Data Corpus

We used the history of SpyWare, to test our benchmark, since it is the largest project we have, with the longest history. In this history, more than 200,000 method calls were inserted, resulting in roughly 200,000 tests for our algorithm, and more than a million individual calls to the completion engine.

We also tested the accuracy of typed completion algorithms by running the benchmark using the type inference engine of Squeak. Only the matches where the type of the object for which completion was attempted were used. This gives us an initial idea of the usefulness of optimist completion in a typed setting.

We also used the data from the 6 student projects, much smaller in nature and lasting a week, to evaluate how the algorithms perform on several code bases, and also how much they can learn in a shorter amount of time. Table 8.1 shows the number of tests for each case study.

Project	Number of Tests
SpyWare	131,000
SpyWare (with types)	49,000
Project A	5,500
Project B	8,500
Project C	10,700
Project D	5,600
Project E	5,700
Project F	9,600

Table 8.1: Number of completion attempts

8.5 Code Completion Algorithms

For each algorithm we present, we first give an intuition of why it should improve the performance of code completion, then describe its principles. We then detail its overall performance on our larger case study, SpyWare. After a brief analysis, we finally provide the global accuracy score for the algorithm, computed from the results. We discuss all the algorithms and their performances on the six other projects in the last section.

8.5.1 Default Untyped Strategy

Intuition: The match we are looking for can be anywhere in the system.

Algorithm: The algorithm searches through all methods defined in the system matching the prefix on which the completion is attempted. It sorts the list alphabetically.

Prefix	2	3	4	5	6	7	8
% 1st	0.0	0.33	2.39	3.09	0.0	0.03	0.13
% 2nd	2.89	10.79	14.35	19.37	16.39	23.99	19.77
% 3nd	0.7	5.01	8.46	14.39	14.73	23.53	26.88
% 4-10	6.74	17.63	24.52	23.9	39.18	36.51	41.66
% fail	89.63	66.2	50.24	39.22	29.67	15.9	11.53

Table 8.2: Results for the default algorithm

Results: The algorithm's score is **12.1**. The algorithm barely, if ever, places the correct match in the top position. However it performs better for the second and third places, which rise steadily: They contain the right match nearly half of the time with a prefix length of 7 or 8, however a prefix length of eight is already long.

8.5.2 Default Typed Strategy

Intuition: The match is one of the methods defined in the hierarchy of the class of the receiver.

Algorithm: The algorithm searches through all the methods defined in the class hierarchy of the receiver, as inferred by the completion engine.

Prefix	2	3	4	5	6	7	8
% 1st	31.07	36.96	39.14	41.67	50.26	51.46	52.84
% 2nd	10.11	11.41	13.84	16.78	13.13	13.51	12.15
% 3nd	5.19	5.94	4.91	5.15	3.2	1.94	2.0
% 4-10	16.29	12.54	12.24	8.12	6.29	4.14	2.79
% fail	37.3	33.11	29.83	28.24	27.08	28.91	30.18

Table 8.3: Results for the default typed completion

Results: The score is **47.95**. Only the results where the type inference engine found a type were considered. This test was only run on the SpyWare case study as technical reasons prevented us to make the type inference engine work properly for the other case studies. The algorithm consistently achieves more than 30% of matches in the first position, which is much better than the untyped case. On short prefixes, it still has less than 50% of chances to get the right match in the top 3 positions.

8.5.3 Optimist Structure

Intuition: Local methods are called more often than distant ones (*i.e.,* in other packages).

Algorithm: The algorithm searches first in the methods of the current class, then in its package, and finally in the entire system.

Prefix	2	3	4	5	6	7	8
% 1st	12.7	22.45	24.93	27.32	33.46	39.5	40.18
% 2nd	5.94	13.21	18.09	21.24	20.52	18.15	22.4
% 3nd	3.26	5.27	6.24	7.22	10.69	14.72	10.77
% 4-10	14.86	16.78	18.02	17.93	17.23	20.51	20.75
% fail	63.2	42.26	32.69	26.26	18.07	7.08	5.87

Table 8.4: Results for optimist structure

Results: The algorithm scored **34.16**. This algorithm does not use the history of the system, only its structure, but is still an optimist algorithm since it orders the matches non-alphabetically. This algorithm represents how much one can achieve without using an additional source of information. As we can see, its results are a definite improvement over the default algorithm, since even with only two letters it gets more than 10% of correct matches. There is still room for improvement.

8.5.4 Recently Modified Method Names

Intuition: Programmers are likely to use methods they have just defined or modified.

Algorithm: Instead of ordering all the matches alphabetically, they are ordered by date, with the most recent date being given priority. Upon initialization, the algorithm creates a new dated entry for every method in the system, dated as January 1, 1970. Whenever a method is added or modified, its entry is changed to the current date, making it much more likely to be selected.

Prefix	2	3	4	5	6	7	8
% 1st	16.73	23.81	25.87	28.34	33.38	41.07	41.15
% 2nd	6.53	12.99	17.41	19.3	18.23	16.37	21.31
% 3nd	4.56	6.27	6.83	7.7	11.53	15.58	10.76
% 4-10	15.53	17.0	20.16	20.73	20.34	20.65	21.55
% fail	56.63	39.89	29.7	23.9	16.47	6.3	5.18

Table 8.5: Results for recent method names

Results: The score is **36.57**, so using a little amount of historical information is slightly better than using the structure. The results increase steadily with the length of the prefix, achieving a very good accuracy (nearly 75% in the top three) with longer prefixes. However the results for short prefixes are not as good. In all cases, results for the first position rise steadily from 16 to 40%. This puts this first "optimist" algorithm slightly less than on par with the default typed algorithm, albeit without using type information.

8.5.5 Recently Modified Method Bodies

Intuition: Programmers work with a vocabulary which is larger than the names of the methods they are currently modifying. We need to also consider the methods which are called in the bodies of the methods they have recently visited. This vocabulary evolves, so only the most recent methods are to be considered.

Algorithm: A set of 1000 entries is kept which is considered to be the "working vocabulary" of the programmer. Whenever a method is modified, its name and all the methods which are called in it are added to the working set. All the entries are sorted by date, favoring the most recent entries. The names of recently modified method are further prioritized.

Prefix	2	3	4	5	6	7	8
% 1st	47.04	60.36	65.91	67.03	69.51	72.56	72.82
% 2nd	16.88	15.63	14.24	14.91	14.51	14.04	14.12
% 3nd	8.02	5.42	4.39	4.29	3.83	4.09	4.58
% 4-10	11.25	7.06	6.49	6.64	6.51	5.95	5.64
% fail	16.79	11.49	8.93	7.09	5.6	3.33	2.81

Table 8.6: Results for recently modified bodies

Results: The score is **70.13**. Considering the vocabulary the programmer is currently using yields much better results. With a two-letter prefix, the correct match is in the top 3 in two thirds of the cases. With a six-letter prefix, in two-third of the cases it is the first one, and it is in the top three in 85% of the cases. This level of performance is worthy of an "optimist" algorithm.

8.5.6 Recently Inserted Code

Intuition: The vocabulary taken with the entire methods bodies is too large, as some of the statements included in these bodies are not relevant anymore. Only the most recent inserted statements should be considered.

Algorithm: The algorithm is similar to the previous one. However when a method is modified, we only refresh the vocabulary entries which have been newly inserted in the modified method as well as the name, instead of taking into account every method call. This algorithm makes a more extensive use of the change information we provide.

Prefix	2	3	4	5	6	7	8
% 1st	33.99	52.02	59.66	60.71	63.44	67.13	68.1
% 2nd	15.05	16.4	15.44	16.46	16.38	17.09	16.52
% 3nd	9.29	7.46	5.98	5.64	5.36	4.74	5.45
% 4-10	22.84	11.05	8.53	8.65	8.45	7.23	6.71
% fail	18.79	13.03	10.35	8.5	6.33	3.77	3.17

Table 8.7: Results for recently inserted code

Results: In that case our hypothesis was wrong, since this algorithm is less precise (the score is **62.66**) than the previous one, especially for short prefixes. In all cases, this algorithm still performs better than the typed completion strategy.

8.5.7 Per-Session Vocabulary

Intuition: Programmers have an evolving vocabulary representing their working set. However it changes quickly when they change tasks. In that case they reuse and modify an older vocabulary. It is possible to find that vocabulary when considering the class which is currently changed.

Algorithm: This algorithm fully uses the change information we provide. In this algorithm, a vocabulary (*i.e.,* a set of dated entries) is maintained for each *development session* in the history. A session is a sequence of dated changes separated by at most an hour. If a new change occurs with a delay superior to an hour, a new session is started. In addition to a vocabulary, each session contains a list of classes which were changed (or had methods changed) during it.

When looking for a completion, the class of the current method is looked up. The vocabulary most relevant to that class is the sum of the vocabularies of all the sessions in which the class was modified. These sessions are prioritized over the other.

Prefix	2	3	4	5	6	7	8
% 1st	46.9	61.98	67.82	69.15	72.59	75.61	76.43
% 2nd	16.88	15.96	14.41	15.01	14.24	14.44	13.8
% 3nd	7.97	5.73	4.64	4.3	3.45	3.0	3.4
% 4-10	14.66	8.18	6.5	6.19	5.44	4.53	4.16
% fail	13.56	8.12	6.58	5.32	4.25	2.39	2.17

Table 8.8: Results for per-session vocabulary

Results: This algorithm is the best we found so far –with a score of **71.67**– even if only by 1.5 points. It does so as it reacts more quickly to the developer changing tasks, or moving around in the system. Since this does not happen that often, the results are only marginally better. However when switching tasks the additional accuracy helps. It seems that filtering the history based on the entity in focus (at the class level) is a good fit for an “optimistic” completion algorithm.

8.5.8 Typed Optimist Completion

Intuition: Merging optimist completion and type information should give us the best of both worlds.

Algorithm: This algorithm merges two previously seen algorithms. It uses the data from the session-based algorithm (our best optimist algorithm so far), and merges it with the one from the default typed algorithm. The list of matches for the two algorithms are retrieved ($M_{session}$ and M_{typed}). The matches present in both lists are further prioritized in $M_{session}$, which is returned.

Prefix	2	3	4	5	6	7	8
% 1st	59.65	64.82	70.09	73.49	76.39	79.73	82.09
% 2nd	14.43	14.96	14.1	13.87	13.17	13.09	12.08
% 3nd	4.86	4.64	3.89	3.27	2.92	2.23	1.85
% 4-10	8.71	7.04	5.86	4.58	4.09	3.37	2.5
% fail	12.31	8.51	6.03	4.75	3.4	1.54	1.44

Table 8.9: Results for typed optimist completion

Results: The result is a significant improvement, by 5 points at **76.79** (we ran it on SpyWare only for the same reasons as the default typed algorithm). This algorithm merely reuses the already accurate session information, but makes sure that the matches corresponding to the right type are prioritized. In particular, with a two letter prefix, it gets the first match correctly 60 percents of the times, compared to 30 and 45 for the two individual algorithms.

8.5.9 Discussion of the results

Most of our hypotheses on what helps code completion where correct, except “Recently inserted code”. We expected it to perform better than using the entire method bodies, but were proven wrong. We need to investigate if merging the two strategies (the vocabulary is the entire body, but recently inserted entries are prioritized further), yields any benefits over using only “Recent modified bodies”. On the other hand, using sessions to order the history of the program is still the best algorithm we found, even if by a narrow margin. This algorithm considers only inserted calls during each session, perhaps using the method bodies there could be helpful as well.

When considering the other case studies (Table 8.10), we see that the trends are the same for all the studies, with some variations. Globally, if one algorithm performs better than another for a case study, it tends to do so for all of them. The only exception is the session-aware algorithm, which sometimes perform better, sometimes worse, than using the code of all the methods recently modified, a close second. One reason for this may be that the other case studies have a much shorter history, diminishing the roles of sessions. The algorithm has hence less time to adapt.

Project	SW	S1	S2	S3	S4	S5	S6
Baseline	12.15	11.17	10.72	15.26	14.35	14.69	14.86
Structure	34.15	23.31	26.92	37.37	31.79	36.46	37.72
Names	36.57	30.11	34.69	41.32	29.84	39.80	39.68
Inserted	62.66	75.46	75.87	71.25	69.03	68.79	59.95
Bodies	70.14	82.37	80.94	77.93	79.03	77.76	67.46
Sessions	71.67	79.23	78.95	70.92	77.19	79.56	66.79

Table 8.10: Scores for the untyped algorithms of all projects

Considering type information, we saw that it gives a significant improvement on the default strategy. However, the score obtained by our optimist algorithms –without using any type information– is still better. Further, our optimist algorithms work even in cases where the type inference engine does not infer a type, and hence is more useful globally. Merging the two strategies, *e.g.,* filtering the list of returned matches by an optimist algorithm based on type information, gives even better results.

8.6 Discussion

Systematic Evaluation

Our approach is the only one to our knowledge allowing a systematic, automatic and repeatable evaluation of code completion engines. In addition, the ability to define a benchmark has proven very valuable to the incremental development of the completion algorithms we tested. It is easy to see if a change results in an improvement when this amounts to comparing two numbers.

Completion of Methods Versus Other Entities

The benchmark we defined only takes into account the completion of method calls, and not other program entities. This is because the number of methods is usually the highest. Other entities, such as packages, classes, variables or keywords are less numerous. Hence the number of methods usually dwarfs the number of other entities in the system, and is where efforts should be first focused to get the most improvements.

Typed Versus Untyped Completions

As we have seen in Section 8.3, there are mainly two kinds of completion: Type-sensitive completion, and type-insensitive completion, the latter being the one which needs to be improved most. We used the Squeak IDE to implement our benchmark. As Smalltalk is dynamically typed, this allows us to improve type-insensitive completion. Since Squeak features an inference engine, we were able to test whether our completion algorithms also improves type-sensitive completion, but only with inferred types.

Applicability to Other Programs

We have tested several programs, but can not account for the general validity of our results. However, our results are consistent among the different program we tested. If an algorithm performs better in one, it tends to perform better on the others. To generalize our results, one simply needs to add new development histories to the benchmark's corpus and run it again.

Applicability to Other Languages

Our results are currently valid for Smalltalk only. However, the tests showed that our optimist algorithms perform better than the default algorithm using type inference, even without any type information. Merging the two approaches shows another improvement. An intuitive reason for this is that even if only 5 matches are returned due to the help of typing, the position they occupy is still important. Thus we think our results have some potential for typed object-oriented languages such as Java. In addition, we are confident they could greatly benefit any dynamically typed language, such as Python, Ruby, Erlang, *etc.*

As for the previous discussion point, adding development histories for these languages would confirm or infirm this hypothesis. Depending on the features of the language, this may require modifying the algorithms as well. However, our algorithms make few assumptions about the structure of the system (the only one being the structure-aware algorithm, and to a limited extent the session-aware algorithm), so the modifications should be minimal.

Replication of Mylyn's Completion

Mylyn's task contexts feature a form of code completion prioritizing elements belonging to the task at hand [KM06], which is similar to our approach. We could however not reproduce their algorithm since our recorded information focuses on changes, while theirs focuses on interactions (they also record which entities were changed, but not the change extent). The data we recorded includes interactions only on a smaller period and could thus not be compared with the rest of the data.

Resource Usage

Our benchmark in its current form is resource-intensive. Testing the completion engine several hundred thousands time in a row takes a few hours for each benchmark. We are looking at ways to make this faster.

8.7 Summary

In this chapter, we tackled the problem of improving and evaluating recommender systems through the example of code completion. Recommender systems monitor a programmer's activity and make recommendations that need to be accurate in order not to slow the programmer down. Even if code completion is a tool used by every developer, improvements have been few and far-between as additional data was needed to both improve it and measure the improvement.

Change-based Software Evolution proved to be a valuable asset to evaluate code completion engines, as it records enough information to simulate the usage of a completion engine at any point in the history of the system. This recorded information was used in the definition of a benchmark allowing systematic testing of code completion engines with realistic data.

This shows that recording development histories is a good way to evaluate recommender systems needing expensive human studies, provided that enough information is available to recreate the context needed by the recommender. Recording a system's evolution in a manner that makes it possible to recreate the system's AST as Change-based Software Evolution does is a significant source of information to gather. Other data sources such as navigation information in the IDE are simpler to record and could be added to the data recorded by Change-based Software Evolution. The availability of this data allows the definition of benchmarks which measure recommender systems in a cheap, accurate and repeatable fashion.

In addition, the data provided by Change-based Software Evolution was also useful to improve the accuracy of the recommendations a completion engine offers. Incorporating change information in the code completion algorithm improved the score from 12 out of 100 to more than 70. This translated to slightly less than a 75% chance of having the match one was looking for with a two-letter prefix. This is due because recent changes are a good approximation of the entities that constitute the working set a programmer is using at any time. Of course, whether this applies to other types of recommender systems in the same proportions remain to be determined.

Chapter 9

Improving Recommendations for Change Prediction

Change prediction consists in recommending artifacts that have a high probability of changing when a given artifact changes. Change prediction is useful both to recommend artifacts whose modification is not obvious, or as a productivity tool to facilitate navigation to entities that the programmer has to modify. If the first can be reliably tested up to a certain extent with SCM archives, there is no way to repeatedly test the second.

The fine-grained granularity of the data we record allows us to replay exact development sessions. With this fine-grained recorded history, we defined a benchmark for change prediction algorithms in order to reliably measure the accuracy of change recommender systems on the entire development history of several projects. With this benchmark, we evaluated several change recommender systems either found in the literature or that we introduced.

9.1 Introduction

In the previous chapter, we investigated the usefulness of Change-based Software Evolution in the case of recommender systems. We showed that using Change-based Software Evolution's data, one can build a comprehensive benchmark to evaluate a specific type of recommender system, code completion. The benchmark being fully automatized allows us to repeat the experiment cheaply, enabling the reproduction of other approaches, their comparison and measuring incremental improvements to recommendation algorithms. We have also shown that the evolutionary data provided by Change-based Software Evolution significantly improved the accuracy of the recommendations of the completion engine.

In this chapter, we bring further support to these conclusions by defining and using a benchmark for change prediction. The difference with the previous chapter is that benchmarks for change prediction tools already exist using SCM archives. The intent of this chapter is hence to show that benchmarks based on Change-based Software Evolution improve on SCM-based ones in the following ways:

- Benchmarks defined by Change-based Software Evolution approximate the behavior of a developer better, as they record and replay actual developer interactions, thus providing a more realistic setting.
- Our benchmarks provide more data, of a more precise nature than SCM-based benchmarks. This in turn makes recommender systems for Change-based Software Evolution outperform those relying on SCM data.
- Benchmarks defined by Change-based Software Evolution can be used to evaluate another class of recommender systems (based on IDE monitoring) in the same unified framework.

The recommender system under study in this chapter is change prediction. Change predictors assist developers and maintainers by recommending entities that may need to be modified alongside the entities currently being changed. Depending on the development phase, change prediction has different usages.

For *software maintenance*, change predictors recommend changes to entities that may not be obvious [ZWDZ04; YMNCC04]. In a software system, there often exist implicit or indirect relationships between entities [GHJ98]. If one of the entities in the relationship is changed, but not the other, subtle bugs may appear that are hard to track down.

For *forward engineering*, change predictors serve as productivity enhancing tools that ease the navigation to the entities that are going to be changed next. In this scenario, a change predictor maintains and proposes a set of entities of interest to help developers focus the programming tasks.

So far, maintenance-mode change prediction has been validated using the history archives of software systems as an oracle. This is an existing benchmark, which is however ad-hoc. It is in particular unadapted to active development, when the transactions are too large to allow an accurate evaluation. As a consequence, no satisfactory approach has been proposed for tools adapted to the forward engineering use case. Those are assessed through comparative studies involving developers [KM06; SES05], a labor-intensive, error-prone and imprecise process (see Section 8.2 for the costs of human subject studies). An accurate benchmark handling both cases is needed.

We present a unifying benchmark for both kinds of change prediction, based on several fine-grained development histories, recorded with Change-based Software Evolution. Our detailed histories are unaffected by the inaccuracies of large transactions, making it usable for both kind of change predictors. It provides (close to) real life benchmark data without needing to perform comparative studies. Based on the benchmark, we perform a comparative evaluation of several change prediction approaches.

Contributions. The contributions of this chapter are:

- A benchmark for change predictor based on the fine-grained data recorded by Change-based Software Evolution. It is more accurate and more generic than previous SCM-based benchmarks.
- The evaluation of several change prediction approaches, some replicated from the literature, some novel. These approaches are representative of the various branches of change prediction.

Structure of the chapter. Section 9.2 describes various change prediction approaches existing in the literature in the two change prediction styles. Section 9.3 justifies and presents our benchmark for change prediction approaches. Section 9.4 details the approaches we evaluated with our benchmark and presents the benchmark results, which we discuss in Section 9.5, before concluding in Section 9.6.

9.2 Change Prediction Approaches

Several change prediction approaches have been proposed, along three major trends. *Historical* approaches and approaches based on *Impact Analysis* have been evaluated on SCM-based data. *IDE-based* approaches, have on the other hand mostly been evaluated with human subjects.

9.2.1 Historical Approaches

Historical approaches use an SCM system's repository to predict changes, primarily in a maintenance setting.

Zimmerman *et al.* [ZWDZ04] mined the CVS history of several open-source systems to predict software changes using the heuristic that entities that changed together in the past are going to change together in the future. They reported that on some systems, there is a 64% probability that among the three suggestions given by the tool when an entity is changed, one is a location that indeed needs to be changed. Their approach works best with stable systems, where few new features are added. It is indeed impossible to predict new features from the history. Changes were predicted at the class level, but also at the function (or method) level, with better results at the class level.

Ying *et al.* employed a similar approach and mined the history of several open source projects [YMNCC04]. They classified their recommendations by interestingness: A recommendation is obvious if two entities referencing each other are recommended, or surprising if there was no relationships between the changed entity and the recommended one. The analysis was performed at the class level. Sayyad-Shirabad *et al.* also mined the change history of a software system in the same fashion [SLM03], but stayed at the file level.

Girba also detected co-change patterns [Gîr05] at the level of classes instead of file, using his metamodel Hismo. He also qualified co-change patterns, with qualifiers such as Shotgun Surgery, Parallel Inheritance or Parallel Semantics. He also proposed the *Yesterday's Weather* measure [GDL04]. Yesterday's Weather postulates that future changes will take place where the system just changed and measures how much a given system conforms to this postulate.

9.2.2 Impact Analysis Approaches

Impact analysis has been performed using a variety of techniques; we only comment on a few. Briand *et al.* [BWL99] evaluated the effectiveness of coupling measurements to predict ripple effect changes on a system with 90 classes. The results were verified by using the change data in the SCM system over 3 years. One limitation is that the coupling measures were computed only on the first version of the system, as the authors took the assumption that it would not change enough to warrant recomputing the coupling measures for each version. The system was in maintenance mode.

Wilkie and Kitchenham [WK00] performed a similar study on another system, validating change predictions over 130 SCM transactions concerning a system of 114 classes. Forty-four transactions featured ripple changes. Both analyses considered coupling among classes.

Tsantalis *et al.* proposed an alternative change prediction approach, based on a model of design quality. It was validated on two systems. One had 58 classes and 13 versions, while the other had 169 classes and 9 versions [TCS05].

Hassan and Holt proposed a generic evaluation approach based on replaying the development history of projects based on their versioning system archives[HH06]. They compared several change prediction approaches over the history of several large open-source projects, and found that historical approaches have a higher precision and recall than other approaches. Similar to Zimmermann *et al.*, they observed that the GCC project has different results and hypothesized this is due to the project being in maintenance mode.

Kagdi proposed a hybrid approach merging impact analysis techniques with historical techniques [Kag07]. They argued that such an approach provides better results than the two other approaches on their own, but no results have been published.

9.2.3 IDE-based approaches

The goal of short-term, IDE-based prediction approaches is to ease the navigation to entities which are thought to be used next by the programmer. These approaches are based on IDE monitoring and predict changes from development session information rather than from transactions in an SCM system. They can thus better predict changes while new features are being built, as they monitor the creation of the new entities and can thus incorporate them in their predictions.

Mylyn [KM06] maintains a task context consisting of entities recently modified or viewed for each task the programmer defined. It limits the number of entities the IDE displays to the most relevant, easing the navigation and modification of these entities. Mylyn uses a Degree Of Interest (DOI) model, and has been validated by assessing the impact of its usage on the edit ratio of developers, *i.e.,* the proportion of edit events with respect to navigation events in the IDE. It was shown that using Mylyn, developers spent more time editing code, and less time looking for places to edit.

NavTracks [SES05] and Teamtracks [DCR05] both record navigation events to ease navigation of future users, and are geared towards maintenance activities. Teamtracks also features a DOI model. NavTrack's recommendations are at the file level. Teamtracks was validated with user studies, while NavTracks was validated both with a user study and also by recording the navigation of users and evaluating how often NavTracks would correctly predict their navigation paths (around 35% of the recommendations were correct).

9.3 A Benchmark for Change Prediction

As with our previous benchmark descriptions, we first motivate the need for a change prediction benchmark, then describe its procedure, how approaches are evaluated and how the results are presented. Finally, we present the change histories our benchmark uses.

9.3.1 Motivation

In Section 9.2 we have listed a number of change prediction approaches that have been evaluated with data obtained either by mining the repositories of SCM systems or with human subject studies. In Section 8.2, we already showed why a human subject study may not be the most adequate validation method. SCM-based benchmarks also suffer from drawbacks.

An SCM-based benchmark proceeds as follows: For each transaction, the set of entities that have changed are extracted. This set is split in two parts, A and B. The predictor is given the set of entities A, and its task is to guess the entities that are in B. The predictor's accuracy can then be measured in terms of precision and recall. This approach however suffers from several limitations.

The first is that the data obtained by mining an SCM repository is potentially inaccurate: Some SCM transactions represent patch applications rather than developments and cannot be used for change prediction. Other transactions are simply too large to extract useful information. SCM transactions in the case of active development are larger, and inherently more noisy as the relationship between two related entities is obscured by other entities in the transaction which are less related. This is one of the reasons why change prediction does not work as well in the case of active development.

Another reason is the arbitrary way in which a transaction is split in two sets of entities. In an SCM transaction, the order of the changes is lost, hence there is no indication of which entities were modified first in the session. These entities would more naturally fit in the set of entities given as context to the predictor. This problem is compounded with the above problem of large transactions.

If these drawbacks are merely a nuisance for history and coupling-based approaches on maintenance systems, they render the SCM-based benchmark approach useless for IDE-based recommender system, and more generally in the context of active development. Active development is characterized by a greater amount of changes, causing larger commits and more noise overall. IDE-based approaches require a finer context, which is simply not possible to reconstruct only from the outcome of the session.

Zeller's vision is that future IDEs are bound to offer more and more assistance to developers in the IDE itself [Zel07]. Evaluating this kind of IDE-based recommender systems needs to be done in a close to real-world setting. SCM-based benchmarks are simply not accurate enough for that.

9.3.2 Procedure

Our benchmark functions similarly to a SCM-based benchmark, but at a much finer level. During each run, we test the accuracy of a change prediction algorithm over a program's history, by processing each change in turn. We first ask the algorithm for its guess of what will change next. The algorithm returns a list of entities expected to change. We evaluate that guess compared to the next changes, and then provide that change to the algorithm, so that it can update its representation of the program's state and its evolution.

Some changes are directly submitted to the prediction engine without asking it to guess the changes first. They are still processed, since the engine must have an accurate representation of the program. These are (1) changes that create new entities, since one cannot predict anything for them, (2) repeated changes, *i.e.,* if a change affects the same entity than the previous one, it is skipped, and (3) refactorings or other automated code transformations.

The pseudo-code of our algorithm is shown in Algorithm 4. It runs on two levels at once, asking the predictor to predict the next changing class and the next changing method. When a method changes, the predictor is first asked to guess the class the method belongs to. Then it is asked to guess the actual method. When a class definition changes, the predictor is only tested at the class level. This algorithm does not evaluate the results, but merely stores them along with the actual next changing entities.

```
Input: $History$: Change history used
$Predictor$: Change predictor to test
Output: $Results$: benchmark results
$Results$ = makeResultSet();
foreach $Session\ S\ in\ ChangeHistory$ do
    storeSessionInfo($S$, $result$);
    $testableChanges$ = filterTestableChanges($S$);  foreach $Change\ ch\ in\ S$ do
        if $includes(testableChanges,ch)$ then
            $predictions$ = predict($Predictor$);
            $nbPred$ = size($predictions$);
            $oracle$ = nextElements($testableChanges$, $nbPred$);
            storeResult($results$, $predictions$, $oracle$);
        end
        processChange($Predictor$, $ch$);
    end
    return $Results$
end
```

Algorithm 4: Benchmark result collection

9.3.3 Evaluation

With these results stored, we can evaluate them in a variety of ways. All of them share the same performance measurement, but applied to a different set of predictions. Given a set of predictions, we use Algorithm 5 to return an accuracy score.

Given a list of n predictions, the algorithm compares them to the next n entities that changed, and sets the accuracy of the algorithm as the fraction of correct predictions over the number of predictions. In the case where less than n entities changed afterwards (m), only the m first predictions are taken into account.

Accuracy vs. Prediction and Recall. Change prediction approaches that use SCM data are often evaluated using precision and recall. We found however that our data does not fit naturally with such measurements, because while recorded changes are sequential, some development actions can be performed in any order: If a developer has to change three methods A, B, and C, he can do so in any order he wants. To account for this parallelism, we do not just test for the prediction of the next change, but for the immediate sequence changes with length n. Defining precision and recall for the prediction set and the set of the actual changes would make both measures have the same value. This does not fit the normal precision and recall measures which vary in inverse proportion to each other. Intuitively, approaches with a high recall tend to make a greater number of predictions, while approaches with a higher precision make less predictions, which are more accurate. Since we fix the number of predictions to a certain size n, we use one single accuracy measure.

We measure the following types of accuracy:

```
Input: $Results$: Benchmark results
$Depth$: Number of predictions to evaluate
Output: $Score$: Accuracy Score

$accuracy = 0$;
$attempts =$ size($results$);
foreach $Attempt$ $att$ $in$ $attemps$ do
    $predictions =$ getPredictions($att, Depth$);
    $oracles =$ getOracles($oracles, Depth$);
    $predicted = predictions \cap oracles$;
    $accuracy = accuracy +$ (size($predicted$) / size($predictions$));
end
$Score = accuracy$ / $attempts$;
return $Score$
```

Algorithm 5: Benchmark result evaluation

- Coarse-grained accuracy (C) measures the ability to predict the classes where changes will occur.
- Fine-grained accuracy (M) measures the ability to predict the methods where changes will occur.
- Initial accuracy (I) measures how well a predictor adapts itself to a changing context both for classes and methods. To evaluate how fast a change predictor reacts to a changing context, we measure its accuracy on the first changes of each session. These feature the highest probability that a new feature is started or continued. We measure the accuracy for the first 20 changes of each session.

9.3.4 Result Format

In the next section we measure the accuracy of a number of approaches using the previously presented benchmark. We present the results in tables following the format of the sample Table 9.1.

For each of the projects (occupying the rows) we compute the coarse-grained (C5, C7, C9), the fine-grained (M5, M7, M9) and the initial accuracy for classes (CI7) and for methods (MI7). The digits (5,7,9) indicate the length of the prediction and validation set, *e.g.,* M7 means that we measure the accuracy of change prediction for a sequence of method changes of length 7.

How are the numbers to be understood? For example, in the C5 column we measure the coarse-grained accuracy of one of the approaches. The '13' in the SpyWare (SW) row means that when it comes to predicting the next 5 classes that will be changed, the fictive

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	13.0	15.5	17.5	2.7	2.9	3.1	17.2	4.1
A-F	27.3	32.0	37.1	3.3	3.7	4.0	34.7	5.5
SA	16.1	19.3	21.9	4.5	5.4	6.2	21.7	6.4
X	6.0	7.4	8.4	2.1	2.2	2.2	7.9	3.7
AVG	14.4	17.2	19.6	3.1	3.5	3.9	18.5	4.6

Table 9.1: Sample results for an algorithm

predictor evaluated in Table 9.1 is guessing correctly in 13% of the cases. In the case of the small student projects (row A-F) the values indicate how the predictors perform on very small sets of data. We aggregated the results of all the students projects in a single row to ease reading. The Software Animator (SA) row indicates how the predictors perform on Java systems, while the second last row (X) indicates how the predictors perform on a system built around a large web framework. The bottom row contains the weighted average value accuracy for each type of prediction. The average is weighted with the number of changes of each of the benchmark systems. This means that the longest history, the one of SpyWare, plays a major role, and that one should not expect arithmetic averages in the last row.

9.3.5 Data Corpus

We selected the following development histories as benchmark data:

- SpyWare, our prototype, monitored over a period of three years, constitutes the largest data set. The system has currently around 25,000 lines of code in 700 classes. We recorded close to 25,000 changes so far.
- A Java project developed over 3 months, the Software Animator. In this case, we used our Java implementation of SpyWare, an Eclipse plugin called EclipseEye[Sha07], which however does not support the recording of usage histories.
- Six one-week small student projects with sizes ranging from 15 to 40 classes, with which we tested the accuracy of approaches on limited data. These development histories test whether an approach can adapt quickly at the beginning of a fast-evolving project.
- A professional Smalltalk project tracked for 3 months, built on top of a web application development framework.

The characteristics of each project are detailed in Table 9.2, namely the duration and size of each project (in term of classes, methods, number of changes and sessions), as well as the number of times the predictor was tested for each project, in four categories: overall class

Project	Days	Sessions	Changes	Classes	Predictions	Early	Methods	Predictions	Early
SpyWare	1,095	496	23,227	697	6,966	4,937	7,243	12,937	6,246
Animator	62	133	15,723	605	3,229	1,784	1,682	8,867	2,249
Project X	98	125	5,513	498	2,100	1,424	2,280	3,981	1,743
Project A	7	17	903	17	259	126	228	670	236
Project B	7	19	1,595	35	524	210	340	1,174	298
Project C	8	19	757	20	215	151	260	538	251
Project D	8	17	511	15	137	122	142	296	156
Project E	7	22	597	10	175	148	159	376	238
Project F	7	22	1,326	50	425	258	454	946	369
Total	1,299	870	50,152	1,947	14,030	9,160	12,788	29,785	11,786

Table 9.2: Development histories in the benchmark.

prediction, overall method prediction, and class and method prediction at the start of sessions only (this last point is explained in Section 9.3.3).

9.4 Results

In this section we detail the evaluation of a number of change prediction approaches using our benchmark. We reproduced approaches presented in the literature and evaluated novel ones. In the case of reproduced approaches, we mention the possible limitations of our reproduction and the assumptions we make. This is followed by the results of each approach and a brief discussion.

We start with a few general remarks. First, the larger the number of matches considered, the higher the accuracy. This is not surprising. One must however limit the number of entities proposed, since proposing too many entities is useless. One can always have 100% accuracy by proposing all the entities in the project. This is why we limited ourselves to 7+2 entities, thus keeping a shortlist of entities, these having still a reasonable probability of being the next ones. This number is estimated to be the number of items that humans can keep in short-term memory [Pin99].

Second, all the algorithms follow roughly the same trends across projects. The smaller projects (A-F) have a higher accuracy, which is to be expected since there are less entities to choose from, hence a higher probability to pick the correct ones. The software animator (SA) project has a higher accuracy than SpyWare (SW), since it is also smaller. The project with the least accuracy overall is project X. Its development is constituted of a variety of smaller tasks and features frequent switching between these tasks. These parts are loosely related, hence the history of the project is only partially useful at any given point in time. Further, project X was already started when we starting monitoring it, so we do not have the full history. Predictions relying on the further past (such as Association Rules Mining) are at a disadvantage.

9.4.1 Association Rules Mining

Description. This is the approach employed by Zimmermann *et al.* [ZWDZ04]. Like Zimmermann's approach, our version supports incremental updating of the dataset to better fit incremental development. The alternative would be to analyze all the history at once, using two-thirds of the data as a training set to predict the other third. This does however not fit a real-life setting. As the approach uses SCM transactions, we make the assumption that one session corresponds to one commit in the versioning system.

When processing each change in the session, it is added to the transaction that is being built. When looking for association rules, we use the context of the 5 preceding changes. We mine for rules with 1 to 5 antecedent entities, and return the ones with the highest support in the previous transactions in the history. Like in Zimmermann's approach, we only look for single-consequent rules.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	13.0	15.5	17.5	2.7	2.9	3.1	17.2	4.1
A-F	27.3	32.0	37.1	3.3	3.7	4.0	34.7	5.5
SA	16.1	19.3	21.9	4.5	5.4	6.2	21.7	6.4
X	6.0	7.4	8.4	2.1	2.2	2.2	7.9	3.7
AVG	14.4	17.2	19.6	3.1	3.5	3.9	18.5	4.6

Table 9.3: Results for Association Rules Mining

Results. Association rule mining serves as our baseline. As Table 9.3 shows, the results are relatively accurate for class-level predictions, but much lower for method-level predictions. The results are in the range of those reported by Zimmermann *et al.* [ZWDZ04]. They cite that for case studies under active development, the precision of their method was only around 4%. Our results for method-level accuracy are in the low single-digit range as well.

9.4.2 Enhanced Association Rule Mining

Description. The main drawback of association rule mining is that it does not take into account changes in the current session. If entities are created during it, as is the case during active development, prediction based on previous transactions is impossible. To address this, we incrementally build a transaction containing the changes in the current session and mine it as well as the previous transactions.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	30.0	36.1	40.8	13.7	16.2	18.4	40.0	23.1
A-F	39.7	45.6	52.2	12.4	14.9	16.6	50.6	23.4
SA	28.0	34.1	39.5	14.2	17.7	20.9	39.4	24.0
X	24.4	29.6	33.7	14.3	17.3	20.2	31.5	31.1
AVG	29.9	35.8	40.8	13.7	16.6	19.0	39.7	24.5

Table 9.4: Results for Enhanced Association Rules Mining

Results. The results of this simple addition are shown in Table 9.4. The prediction accuracy at the class-level is higher, but the method-level accuracy is much higher. Incrementally building the current session, and mining it allows us to quickly incorporate new entities which have been created in the current session, something that the original approach of Zimmermann does not support, because the data is not available. Of note, the algorithm is more precise at the beginning of the session than at the end, because the current session has less entities to propose at the beginning of the session. Towards the end of the session, there are more possible proposals, hence the approach loses some of its accuracy. In the following, we compare other approaches with enhanced association rule mining, as it is a fairer comparison since it takes into account entities created during the session.

9.4.3 Degree of Interest

Description. Mylyn maintains a degree-of-interest model [KM05] for entities which have been recently changed and edited. We implemented the same algorithm, with the following limitations:

- The original algorithm takes into account navigation data in addition to change data. Since we have recorded navigation data only on a fraction of the history, we do not consider it. We make the assumption that navigation data is not essential in predicting future change. Of course, one will probably navigate to the entity he wants to change before changing it, but recommending to change what one is currently looking at is hardly a useful recommendation.

- Another limitation is that more recent versions of the algorithm [KM06] maintain several degrees of interests based on manually delimited tasks. The tasks are then recalled by the developer. We do not consider separate tasks. The closest approximation of that for us would be to assume that a task corresponds to a session, maintain a degree-of-interest model for each session, and reuse the one most related to the entity at hand.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	16.1	21.0	25.7	10.2	12.8	14.8	20.1	12.5
A-F	52.0	60.2	67.0	16.6	20.7	23.3	57.8	20.2
SA	22.4	29.8	35.5	21.2	26.9	31.0	29.7	25.2
X	15.5	19.5	22.0	6.1	7.4	8.2	17.1	6.5
AVG	21.9	27.6	32.5	13.2	16.6	19.1	25.7	14.9

Table 9.5: Results for Degree of Interest

Results. We expected the degree of interest to perform quite well and found the results to be below our expectations. At the class level, it is less precise than association-rule mining. At the method level, it has roughly the same accuracy. The accuracy drops sharply with project X, whose development involved continuous task switching. Since the degree-of-interest needs time to adapt to changing conditions, such sharp changes lowers its performance. Indeed, the best accuracy is attained when the algorithm has more time to adapt. One indicator of this is that the algorithm's accuracy at the beginning of the session is lower than the average accuracy. The algorithm also performs well to predict classes in projects A-F, since their size is limited. It nevertheless shows limitations on the same projects to predict methods given the very short-term nature of the projects (only a week).

9.4.4 Coupling-based

Description. Briand *et al.* found that several coupling measures were good predictors of changes [BWL99]. We chose to run our measure with the PIM coupling metric, one of the best predictors found by Briand *et al.* PIM is a count of the number of methods invocations of an entity A to an entity B. In the case of classes, this measure is aggregated between all the methods in the corresponding two classes.

To work well, the coupling-based approach needs to have access to all the code base of the system and the call relationships in it. With respect to this requirement, our benchmark suffers from a number of limitations:

- We could not run this test on project SA, since our Java parser is currently not fine-grained enough to parse method calls.
- We also do not include the results of Project X. Unfortunately, project X was already under development when we started recording its evolution, and relies on an external framework. Since we could not access that data, including the results would make for an unfair comparison.
- One last limitation is that the Smalltalk systems do not have any type information, due to the dynamically typed nature of Smalltalk. This makes our PIM measure slightly less accurate.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	21.0	26.2	30.6	9.7	11.8	13.3	25.3	11.2
A-F	21.3	28.0	34.0	10.8	14.1	16.6	29.9	15.7
SA	-	-	-	-	-	-	-	-
X	-	-	-	-	-	-	-	-
AVG	21.0	26.6	31.3	9.9	12.3	14.0	26.1	12.1

Table 9.6: Results for Coupling with PIM

Results. As we see in Table 9.6, comparing with the other approaches is difficult since part of the data is missing. On the available data, we see that the coupling approach performs worse at the method level than Association Rules Mining and Degree of Interest. At the class level, results are less clear: The approach performs worse than association rules, while the degree of interest performs significantly better on projects A-F, but is outperformed at the class level.

Overall, we share the conclusions of Hassan and Holt [HH06]: Coupling approaches have a lower accuracy than history-based ones such as association rule mining. The comparison with degree of interest somewhat confirms this, although it was outperformed in one instance.

9.4.5 Association Rules with Time Coupling

Description. In previous work [RPL08] we observed that a fine-grained measure of logical coupling was able to predict logical coupling with less data. We therefore used one of these fine-grained measurements instead of the classical one to see if it was able to better predict changes with association-rule mining. When mining for rules, instead of simply counting the occurrences of each rule in the history, we factor a measure of time coupling. Time coupling measures how closely in a session two entities changed. Entities changing very closely together will have a higher time coupling value than two entities changing at the beginning and at the end of a development session.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	30.1	36.0	40.6	13.2	16.0	18.3	38.0	21.4
A-F	37.0	43.6	50.8	14.6	17.3	19.3	45.7	23.3
SA	25.6	31.0	35.7	17.2	20.7	23.7	33.0	23.1
X	23.8	29.0	33.2	10.8	14.6	17.6	29.9	25.9
AVG	29.0	34.7	39.7	14.0	17.2	19.7	36.6	22.6

Table 9.7: Results for Association Rules with Time Coupling

Results. As we see in Table 9.7, our results are mixed. The prediction accuracy is slightly lower that Enhanced Association Rules Mining for class-level predictions, and slightly better for method-level predictions, each time by around one percentage point. It is encouraging that the method prediction is increased, since it is arguably the most useful measure: Precise indications are better than coarser ones. A possible improvement would be to use the combined coupling metric we defined, which was more accurate than the time coupling alone.

9.4.6 HITS

Description. Web search engines treat the web as a graph and feature very impressive results. Examples of search algorithms on the web are the HITS algorithm [Kle99] and Google's PageRank [PBMW98]. HITS gives a *hub* and a *sink* value to all nodes in a directed graph. Good *hub* nodes point to many good *sink* nodes, and good *sinks* are referred to by many good *hubs*. The algorithm starts with an initial value for each node which is adjusted at each iteration of the algorithms, depending on the hub and sink values of its neighbors.

While the original HITS works on hyperlinked web pages, we build a graph from classes and methods, linking them according to containment and message sends. The predictor has two variants, respectively recommending the best hubs or the best sinks. Hubs are classes or methods pointing to a large number of other classes or methods, while sinks are classes or methods pointed at by a large number of other entities. To account for recent changes, we maintain this graph for the entities and methods defined or changed in the 50 last changes.

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	40.8	48.8	55.8	15.9	17.1	17.9	48.3	17.2
A-F	44.7	55.7	65.1	20.9	23.5	25.2	57.6	22.6
SA	55.4	71.5	85.7	31.3	35.6	38.9	73.6	33.0
X	30.4	33.7	36.7	7.7	8.4	8.8	32.4	8.6
AVG	43.1	52.6	61.0	19.2	21.4	22.8	51.8	19.6

Table 9.8: Results for Hits, best hubs

Project	C5	C7	C9	M5	M7	M9	CI7	MI7
SW	50.0	55.9	61.1	16.7	17.9	18.6	55.8	17.7
A-F	57.6	66.0	74.6	22.3	24.9	26.7	65.8	24.0
SA	60.6	74.6	87.5	32.1	36.2	39.0	76.2	33.5
X	34.0	38.1	40.8	8.0	8.6	9.0	37.3	8.5
AVG	51.0	58.8	65.8	20.1	22.1	23.4	58.0	20.1

Table 9.9: Results for Hits, best sinks

Results. The HITS algorithm proved to be the highest performer overall since it has a significantly higher method-level accuracy overall. We tested two variants, the first one returning the best hubs in the HITS graph, and the second returning the best sinks. Of the two, recommending sinks seems to consistently achieve a higher accuracy than recommending hubs. As with the degree-of-interest, HITS tends to be more precise towards the end of a session. We need to investigate ways to make the algorithm adapt faster to new contexts.

9.4.7 Merging Approaches

Description. Kagdi *et al.* advocated merging history based approaches and impact analysis based approaches [Kag07], arguing that combining the strong points of several approaches can yield even more benefits. This strategy was successfully used by Poshyvanyk *et al.* in a related problem, feature location [PGM+07]. We tried the following merging strategies:

Top: Returning the top picks of the combined approaches. This approaches assumes that the first choices of an approach are the most probable ones. Given two predictors A and B, it returns A's first pick, then B's first pick, then A's second pick, *etc.*.

Common: Returning entities nominated by several approaches, followed by the remaining picks from the first approach. This approach assumes that the first predictor is the most accurate, and that the other predictors are supporting it. The approach returns all the predictions in common between at least two predictors, followed by the remaining predictions of the first predictor.

Rank addition: This approach favors low ranks in the result lists and entities nominated by several predictors. An interest value is computed for each prediction. Each time the prediction is encountered, its interest is increased by a value proportional to its rank in the prediction list it is in. If it is encountered several times, its interest value will thus be higher. The predictions are then sorted by their interest value.

Predictor 1	Predictor 2	Strategy	Score	Increase
Hits-Sinks	Coupling	Common	40.6	+0.1
Hits-Hubs	Interest	Common	37.8	+0.8
Hits-Hubs	Coupling	Common	37.3	+0.3
Rule Mining	Interest	Common	27.4	+1.2
Rule Mining	Time Coupling	Top	27.3	+1.1
Rule Mining	Coupling	Rank	26.3	+0.1
Time Coupling	Coupling	Rank	26.1	+0.2
Interest	Coupling	Rank	22.6	+0.5

Table 9.10: Results when merging two prediction approaches

Results. We tried these merging approaches on each possible combination of two and three predictors. In the following table we report on the successful ones, where the accuracy of the merging was greater than the best of the merged approaches. If several merging strategies were successful with the same pair of predictors, we only report the best performing one. We only report one accuracy figure, which is the average of C7 and M7. The results are shown in Table 9.10.

We see that merging is successful in some cases, although the effect is limited. The predictions based on the HITS algorithm see some improvement only with the "Common" strategy, which favors the first predictor. This is to be expected, since these algorithms are significantly more accurate than the other ones. Systematically merging their top candidates with the ones of a less accurate approach automatically decreases their efficiency.

With other predictors, merging strategies becomes more viable. The "Top" strategy is the less successful, as it appears only once. This strategy is the only one not rewarding the presence of common recommendations in the two predictors. In the case where "Top" yielded an improvement of 1.1%, using "Rank" instead gave a slightly smaller increase of 0.9%. Overall, merging strategies favoring common predictions work best, since "Rank" appears three times, and "Common" four.

Perhaps the most important fact is that Coupling appears in five of the eight successful mergings. This supports the idea that coupling based on the structure of the system proposes different predictions than history-based ones. Our result provide initial support to Kagdi's proposition of merging impact-analysis approaches (some of them using coupling measurements) with history-based approaches.

Surprisingly, merging three predictors instead of two yielded few benefits. Only in one instance was it better than merging two. Merging Association Rule Mining, Degree of Interest and Coupling had a score of 27.9, a 0.5% increase over merging only Association Rule Mining and Degree of Interest. Of note, only Rank and Common were successful in merging three predictors. The higher the number of predictors, the more important it is to favor common predictions.

9.4.8 Discussion of the results

The results of all the predictors are summed up in Table 9.11, and graphically in Figure 9.1. Note that the coupling results were not run on all the projects, and should as such be taken with a grain of salt. The last two columns represent an overview value for each predictor: O7 is the average of C7 and M7, while MO7 is another average of C7 and M7, favoring method accuracy (C7 counts for a third and M7 for the two remaining thirds).

Predictor	C5	C7	C9	M5	M7	M9	CI7	MI7	O7	MO7
Association Rules Mining	14.4	17.2	19.6	3.1	3.5	3.9	18.5	4.6	10.35	8.06
Enhanced Rules Mining	29.9	35.8	40.8	13.7	16.6	19.0	39.7	24.5	26.20	23.00
Degree of Interest	21.9	27.6	32.5	13.2	16.6	19.1	25.7	14.9	22.10	20.26
Coupling-based*	21.0	26.6	31.3	9.9	12.3	14.0	26.1	12.1	19.45	17.06
Rules Mining & Time Coupling	29.0	34.7	39.7	14.0	17.2	19.7	36.6	22.6	25.95	23.03
Hits, best Hubs	43.1	52.6	61.0	19.2	21.4	22.8	51.8	19.6	37.00	31.80
Hits, best Sinks	51.0	58.8	65.8	20.1	22.1	23.4	58.0	20.1	40.45	34.33

Table 9.11: Comprehensive results for each predictor

(Enhanced) Association Rules Mining

Taking into account recent changes to predict the very next changes in the system is important, as shown by the difference between association rules mining and enhanced association rules mining. The only addition to enhanced association rules mining is to also mine for rules in the current session. However the results are drastic, since its accuracy more than doubles.

We tried to alter the importance of the rule based on the changes in the sessions. We found that taking into account the timing of the changes when they occurred in a session decreased the accuracy at the class level, but increased it at the method level. This may be because classes are interacted with on longer portions of a development session, while method interactions are much more localized in time, and usually worked at only for brief periods of time. Hence the measure is much more useful for predicting method changes.

Degree of Interest

Considering only recent changes is sometimes not enough. We expected the degree of interest to perform better than association rules mining. Although their accuracy is comparable, association rules mining is a more accurate prediction approach. This is due both to the adaptation factor of the degree of interest when switching tasks (its initial accuracy is lower), and the fact that the association rules look in the entire past and can thus find old patterns that are still relevant. The Mylyn tool, which uses a degree of interest, has a built-in notion of tasks [KM06], that alleviate these two problems: A degree of interest is maintained for each task, and is manually recalled by the developer. Task switching recalls another degree of interest model, which may also contain information from the further past. Therefore, we need to evaluate the accuracy of combining several degrees of interest and selecting the one best adapted to the task at hand.

Coupling

Coupling based on the system structure is overall less accurate than other approaches. This is to be expected this is does not take into account recent or past changes at all. However, it proved to be an efficient alternative when prediction approaches were combined. Using it as a second opinion significantly raised the accuracy of some approaches.

HITS

Overall, as Figure 9.1 illustrates, the best performing approach we found is the HITS algorithm, using a graph describing the structure of the system among the recent changes (in our case the last 50 changes). The HITS algorithm can be applied to any graph, so alternative definitions of the graph based on the same data may yield even better results. Our graph definition considers both recent changes and some structural information about the system. It is a trade-off between a pure change-based and a pure structure-based approach, which is a reason why it performs well. A possible improvement would be to incorporate change data

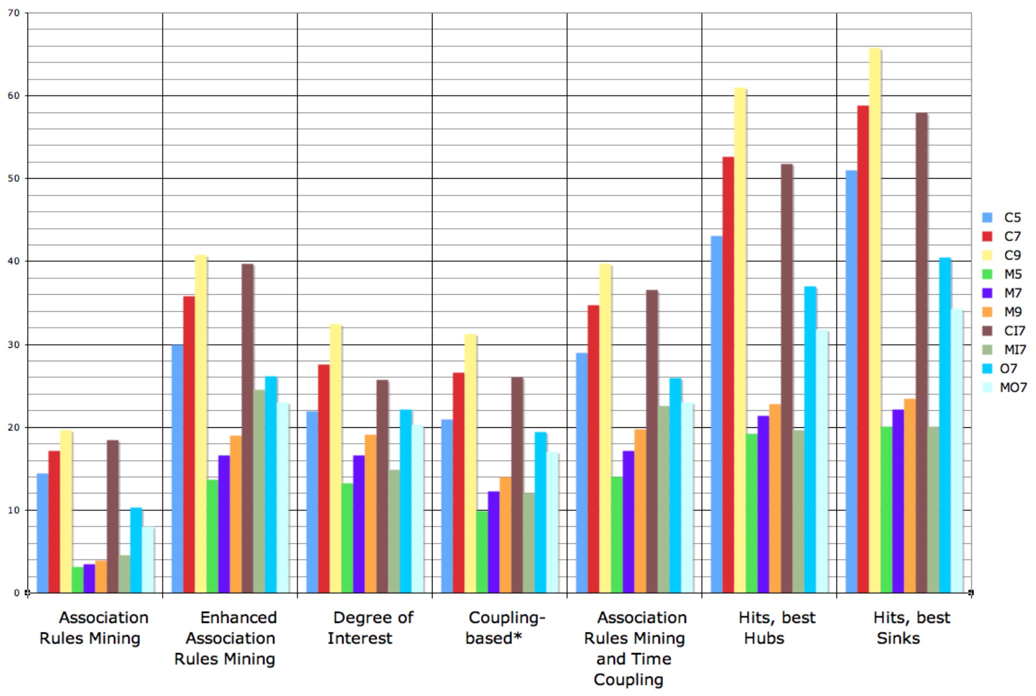


Figure 9.1: Prediction Results

from the further past. Since HITS is –as the Degree of Interest– sensible to task switching, in the future we need to evaluate the accuracy of several HITS graphs combined.

9.5 Discussion

Replication of Approaches in The Literature

We did not reproduce the NavTracks approach as it relies only on navigation data, which we do not have. Ying and Shirabad's approaches are very close to Zimmermann's association rule mining. DeLine *et al.*'s Teamtrack is based on a DOI and is as such close to the Mylyn DOI. Kagdi's approach was not fully described at this time of writing. Finally, we chose only one coupling measure to reproduce, while many exist. The one we chose was the one best adapted to our systems as PIM takes polymorphism into account. In Briand *et al.*'s study, PIM was one of the metrics with the best correlation with actual changes, hence we deem it to be a good representative.

Size of The Dataset

Our dataset is fairly small compared to the ones available with versioning system data. An advantage of the benchmark approach is that it is easy to solve this problem by adding other development histories. On the other hand, our benchmark is already larger than the systems used in previous studies by Briand, Wilkie or Tsantalis. Their evaluations were done on one or two systems, on a small number of transactions.

Generalizability

We do not claim that our results are generalizable. However, some of the results we found were in line with results found by other researchers. Hassan and Holt found that coupling-based approaches are less precise than history based approaches, and so do we. Similarly, Zimmermann *et al.* find a precision of 4% for method change prediction during active development. Reproducing the approach with our data yields comparable results. We also found evidence supporting Kagdi's proposal of merging association rule mining and coupling-based impact analysis. A simple merging strategy performed better than the individual strategies. A more developed merging strategy may produce improved results.

Absent Data

Our data does not include navigation data, which is used in approaches such as NavTracks. Mylyn's Degree of Interest also includes navigation data. We started recording navigation data after recording changes. As such, we only have navigation data for a portion of SpyWare's history. The lack of navigation data needs to be investigated, in order to see if it impacts the accuracy of our implementation of Degree of Interest.

Evolving The Benchmark

Our benchmark still needs to evolve. As said earlier, the dataset should be larger for the results to be more generalizable. Another possible enhancement to the benchmark would be to evaluate if a tool is able to predict a change's *content* and not only its location. Some possibilities would be to evaluate the size of the change, *i.e.,* whether a small or a large change is expected, or the actual content. The latter could be possible and useful to evaluate automatized code clone management approaches.

9.6 Summary

In this chapter, we evaluated the usefulness of Change-based Software Evolution for benchmarking change prediction. Change prediction was previously evaluated by benchmarks, but these relied on data originating from SCM archives. These have limitations: They depend on the size of the transactions to be accurate, while these may be too large and noisy. SCM transaction also lose the order in which changes were performed, an additional source of noise. Thus SCM-based benchmarks are not adapted to usage in active development or in an IDE setting. This prevents change predictors based on IDE monitoring to be evaluated with a benchmarking approach. These relied on human subject studies for their evaluations, which are more expensive to set up and harder to reproduce.

We showed that benchmarks built on top of Change-based Software Evolution have a finer granularity of results thanks to the sequential nature of the changes. This translates in a more realistic setting to test the recommenders. As a result, we could include in the comparison algorithms that relied on IDE monitoring and evaluate them with the same settings as other approaches, allowing a more direct comparison. In total we evaluated and compared seven different approaches to change prediction.

The change information was also useful to improve the accuracy of the recommenders. The most striking example was that including information about the session that is currently being built in the association mining algorithm drastically improved it results, more than doubling its accuracy. This shows that recent usage is a very good indicator of the context a developer is building. Our best performing predictor incorporates recent change information with structural information, using this data to build a graph which it queries using the HITS algorithm.

Finally, our benchmark allowed us to systematically experiment in combining prediction strategies. We came up with conclusions as to what kind of merging strategy is best (strategies emphasizing common results among several predictors), and which change predictor is best combined with other predictors (predictors using very different selection criteria).

Part IV

First-class Changes: So What?

Chapter 10

Perspectives

As software evolution takes up a larger and larger part of the life cycle of software systems, it becomes more and more important to streamline its process. In this dissertation, we argued that representing change as a first-class entity allows us to assist several aspects of a system's evolution, in the context of both reverse and forward engineering. To validate our thesis, we designed and implemented such a change-based model of software evolution. We validated it by applying it to representative tasks in software evolution.

10.1 Contributions

During the course of this dissertation, we made the following major contributions:

10.1.1 Defining Change-based Software Evolution

We argued that current models of software evolution are incomplete, and that the limitations they have can be addressed with a *change-based* model of software evolution.

Gathering requirements. In **Chapter 2** we gathered requirements for a better support of software evolution through an analysis of state of the art software evolution models and approaches. The key shortcomings we identified were that a model of software information should be accurate enough to support fine-grained analyses, be abstractable to support coarse-grained ones, and should not depend on versioning systems, prone to information loss.

Change model design. In **Chapter 3** we designed a change-based model of software evolution fulfilling the requirements we identified. It models the evolution of fine-grained programs (ASTs), through first-class changes which are recorded from an IDE rather than being recovered from SCM archives. The changes our model support are executable, undoable, and composable.

10.1.2 Change-based Software Evolution in Reverse Engineering

Before performing actual changes, developers spend more than half of their time reading and understanding the systems they maintain. We investigated how Change-based Software Evolution supports the understanding of systems at several levels of granularity:

High-level reverse engineering. In **Chapter 4** we showed that fine-grained changes can be abstracted to uncover high-level relationships between entities in a software system. Through a comprehensive visualization of a system's evolution, we identified several visual patterns characterizing the relationships between entities. From this, we demonstrated how one can reconstruct an evolution scenario of the system. We showed that without the fine-grained data provided by Change-based Software Evolution, the quality of the analysis is significantly degraded.

Low-level program comprehension. In **Chapter 5** we showed how understanding a piece of code was eased by following the footsteps of the one who implemented it. Reviewing changes in a development session highlights relationships between entities, orders the changes in a logical way, and contextualizes the changes. To give additional context, we also defined a characterization of sessions based on several change-based metrics qualifying several aspects of a given session. Both session characterization and incremental program understanding are unique to our approach.

Accurate metric definition. Metrics are widely used to summarize a large amount of data. Change-based Software Evolution supports a wide array of metrics, since it supports metrics relying on a fine-grained system representation, and metrics based on a fine-grained change representation. These qualities allowed for the definition of more accurate evolutionary metrics: Our alternative measurements of logical coupling, defined in **Chapter 6**, were able to predict logically coupled entities with less history than the standard definition of logical coupling.

10.1.3 Change-based Software Evolution in Forward Engineering

Beyond understanding systems, maintainers need assistance performing the actual changes to the system. We investigated how Change-based Software Evolution supports automated program transformations and recommender systems:

Program transformation. Program transformations automatize changes that would otherwise be manual and error-prone. In **Chapter 7**, we extended Change-based Software Evolution with program transformations in a natural and unobtrusive way: Program transformations are simply change generators. We also showed how recorded changes can be used as examples to ease the definition of program transformations. The structure of the recorded change itself acts as a checklist of what need to be done to generalize the change in a transformation. Finally, we showed that our definition of program transformations is fully integrated in the evolution: Changes generated to transformations can be traced back to them, easing their comprehension and the evolution of the transformation itself if necessary.

Evaluating and improving recommender systems. Recommender systems assist programmers while they perform changes by indicating where they should focus their attention. They must however be evaluated with care since inaccurate recommenders are harmful to the productivity. Such an evaluation is difficult to do without expensive human subject studies, since recommenders rely on real-world IDE usage. Through two examples, code completion (in **Chapter 8**) and change prediction (in **Chapter 9**), we showed that recording development histories with Change-based Software Evolution led to the definition of robust benchmarks to evaluate recommender systems. When a benchmark based on SCM data previously existed, we showed that the benchmark based on Change-based Software Evolution was more precise and corresponded more to real-world IDE usage. Based on these benchmarks, we evaluated several variants of completion engines and change predictors, and found that those using fine-grained change data performed best.

10.1.4 Additional Contributions

Implementing Change-based Software Evolution. As a technical contribution, we provide an implementation of Change-based Software Evolution in the form of SpyWare for Smalltalk, which we used to perform our evaluations. A proof-of-concept implementation for Java and Eclipse was implemented by a student.

Populating a change-based software repository. We constituted an initial repository of change-based development histories, composed so far of nine case studies: Our prototype itself, monitored over a period of three years, a web-based project monitored over three months, several week-long student projects, and one project monitored with the Java version of Change-based Software Evolution, implemented over three months. All of the case studies were used at one point or another in the validation of Change-based Software Evolution.

10.2 Limitations

Each validation step and its limitations have been discussed to a certain extent in the corresponding chapter. In this section we discuss more general threats to the validity of the work as a whole, or common to several of the validations. After that, we discuss some possible technical threats to general adoption of the work and ideas to deal with them.

10.2.1 Threats to Validity

One major characteristic of using software repositories for evolution analysis is that the data one uses is recorded and analyzed in a postmortem fashion. A further characteristic of Change-based Software Evolution is that the changes are recorded as they happen in the IDE instead of at the end of a task. We discuss threats to validity according to the type of threat with a focus on these two particular aspects.

Internal Validity Internal validity refers to the validity of inferences in an experimental setting. Several of the threats in this category (such as testing, maturation, experimental mortality) are alleviated by the fact that the developer interactions are recorded, sidestepping these threats related to the adaptation of subjects to the experiments.

The largest threat to internal validity we observed is related to *instrumentation*, *i.e.,* the fact that we record interactions may modify the behavior of a developer. Since even changes inducing errors are recorded, developers may be reluctant to perform changes as naturally as they would without the monitoring of the changes, adopting a more careful approach. We made our recording as unobtrusive as possible but some developers may still be wary of it. Hence we cannot guarantee that recording does not affect a developer's behavior.

A possible threat is the usage of our own tool for evolution analysis, as being its implementor would potentially give us an unfair advantage. We do not consider it as a threat for two reasons. First, this would apply only to the case studies related to understanding a system (Chapters 4 and 5). The automated validations are too systematic and undiscriminating for it to make a difference. We only used SpyWare as a case study in Chapter 5, along with another case study, project X. We reported on 3 sessions from project X, and one from SpyWare. Hence a large majority of the case studies did not involve understanding our own system. Second, a developer trying to understand his own past changes is actually a use case of our tool in the case of continuous reverse engineering activities. In such a case, there would be an advantage, but it can hardly be considered unfair.

External Validity Threats to external validity are related to how generalizable the results are beyond the projects they were applied to. Empirical studies of systems are difficult to generalize to other systems and our studies are no exception. Development styles, practices and conditions vary between projects, so a large number of variables can have a potential impact. In this work we intentionally restricted our focus to changes in order to better isolate their effects. Indeed, changes to a developing system may be the least common denominator across a large variety of systems.

Potential threats to the generalizability of our results are that we used a limited number of projects (12) for our analyses. These projects were of small to medium sizes and primarily in active development, making the generalization to large systems and/or in maintenance mode, non trivial.

Further, the projects we considered were single user, and for the most part in a single programming language, Smalltalk. However, one of the projects we monitored was a Java project, indicating that some of our results are at least partly applicable to Java. In addition, we monitored developer with various programming experience (from 6 months to several years) and styles. We found consistent improvements overall, especially in the validations employing benchmarking approaches, for which the results are easier to compare: If an algorithm performed better on one project, it did too on the others.

The best way to address this issue is to replicate our validation experiments on other systems, of different size, programming styles and maintenance life-cycle phases. The original information gathering effort will be rather costly, but once the history of one or more systems of this kind is recorded, the other validations can be replicated at will on their change histories.

Construction Validity Construct validity tests whether the collected data is adequate for the tasks we intend to experiment with.

Since software evolution is primarily about changing existing systems, collecting change information about systems is natural. In addition, the change data we collect is more fine-grained and accurate than existing approaches, giving a more accurate view on the evolution of systems. Since we record the changes through an automated process, data collection is undiscriminating.

One could argue that the level of detail we record is too detailed. Erroneous changes such as unsuccessful attempts by developers are recorded, which would not appear in a versioning system. These changes could be misleading. However, entities that are introduced to be removed during the same development session could be easily filtered out. The effect of such a filtering has to be evaluated as part of our future work.

In parts of our work, we use the assumption that a development session is equivalent to a versioning system commit. This assumption may not be correct. In particular, it is easy to imagine scenarios in which several sessions lead to a commit, or a single session leading to several commits. The more general problem is the recovery of actual tasks. Indeed, SCM commits do not always correspond to development tasks either. Finding how to recover development tasks from the change history is an interesting future work area.

On the other hand, one could also argue that we do not use enough information. Additional information sources such as navigation in the source code, bug archives, mailing lists, *etc.* complement the usage of change information by providing additional points of views on the system's evolution. In this work, we intentionally focused on changes to the system, purposely ignoring other information sources. Since we focused on changes, we wanted to restrict the number of variables to take into account (the development process is complex enough as it is). These additional information sources could be used as part of future work. In fact, SpyWare already records navigation information, but does not use it yet.

Reliability Reliability refers to the ability to consistently measure the effect of the approach. One strength of Change-based Software Evolution is that being based on recorded interactions, it eases the reproducibility of the results.

The changes are systematically recorded and can be reused easily to repeat an experiment. We exploited this to the maximum in Chapters 6, 8 and 9, when we used automated benchmarks to evaluate metrics and recommendation systems. These approaches are very easily repeated. Approaches in Chapter 4, 5 and 7 are more expensive to reproduce and subject to more variability since they involve human interaction. This is especially the case with our example-based approach in Chapter 7 which was evaluated on few individual examples. We consider this exploratory validation the first step in our evaluation of example-based transformation and plan a further evaluation increasing the amount of automation if possible.

Another aspect is replication on other case studies. The first step to this is to gather additional information in the form of new recorded change histories. This step is the hardest since it involves the most user interaction. Once a new history is available, it can conceivably be used for each of the techniques we introduced, increasing the level of confidence for every one of them.

10.2.2 Adoption Issues

We identified several issues that may hamper the widespread usage of Change-based Software Evolution. We describe these problems and propose possible solutions to them.

Technical Issues For now, our system supports only single user development and has been used for small to medium scale projects. To adapt to the realities of industrial development, a multi-user version has to be developed. Tests have to be performed to identify potential bottlenecks related to larger-scale systems.

Beyond this, an important aspect is to reduce the barrier to entry by providing a seamless and unintrusive experience to users, as they will be reluctant to change their habits. We kept this aspect in mind when implementing our prototype and to this aim kept the recording of the changes as unobtrusive as possible. Recording changes while coding causes no interruptions for the developer.

A second way to reduce the barrier to entry is to complement, rather than replace, standard SCM systems. Versioning systems are very slow to be replaced, due to migration issues, process habits, and the general unwillingness to learn a new tool. Versioning systems generally also store more than the system's source code, making a language-specific versioning system a hard sell. Hence we view our system as a layer on top of a standard versioning system rather than a versioning system replacement. Our current change format being very simple, it is easy to store as text files or in a database.

A final concern is performance and space requirements. Currently, the change history of 3 years of SpyWare's development is measured in the low tens of megabytes, making it very reasonable considering that a standard machine's hard drive contains hundreds of gigabytes. Space in memory (when the actual system representation is built) is higher at the moment, but we have done very few optimizations. This is part of the work needed to adapt SpyWare to larger-scale systems. An optimization already done is that it is not necessary to have the entire system's AST in memory all the time. Parts of it can be built from subsets of the change history on request. This considerably reduces runtime memory requirements. Based on this, accessing the state of any entity at any point in time is a matter of seconds.

Ethical Issues There are ethical issues related to monitoring the activity of developers as they work. Developers may be unwilling to be monitored in such a way, and the information could be used to make unethical decisions.

However, the problem exists in a latent form with SCM systems. An SCM system also monitors a developer's changes and activity in the system. The information in an SCM system could be used for the same purposes, and this has not been a major problem for SCM. If anything, the information we record is a more accurate summary of one's activity. As such, it is less error-prone than recovering information from a versioning system and would be more fair to any user of the system. Steps towards ensuring the anonymity of the recorded changes could be imagined. In such a case, one would have to evaluate the trade-off between increased anonymity and decreased usefulness of the information. Recommender evaluation and program comprehension would still be possible, but contacting the responsible of a given change would be harder.

10.2.3 Conclusions

In this work, we focused on a specific kind of evolutionary information, the changes performed on an evolving system in the single-user case. To ensure that this work is valid on larger domains, the studies we performed need to be replicated on several other systems in order to verify if our findings hold in other contexts. Additional sources of information and analyses need to be integrated. To ensure that Change-based Software Evolution has an impact on everyday programmers, it needs to be extended to handle multiple users, ensuring that the technical and ethical issues we described are addressed correctly.

10.3 Lessons Learned

During the four years in which we completed this work, we learned a tremendous amount. Some of the most valuable lessons we learned follow.

On taking a clean slate approach. This decision was the riskiest we took, and in the end the most gratifying. A significant part of our work is based on mining software repositories, where the quality of the data is paramount. Hence our initial decision to discard all the available data seemed at times a choice no sane person would make. This had an impact on the generalization of our results. At the end of these four years, we have a bigger amount of data, but still no large project which could be deemed representative of software evolution in general. Then again, we are not sure any project is.

On the other hand, the clean slate approach allowed us to freely choose the data we needed and as a result gave us higher quality data. We in essence traded quantity for quality. It is now time to consolidate by gathering more data.

On model simplicity and flexibility. The accuracy of our model is directly tied to how domain-specific it is and inversely tied to how adaptable it is to other domains. To minimize portability issues, we kept the core data definitions as generic as possible and defined domain-specific problems as extensions/

We think we succeeded in this aspect since we managed to implement an initial version of our model for Java, with which we recorded the evolution of a project over several months. This does not mean that porting the approach to a new platform is easy. There will always be a minimal cost to achieve this.

On multiple validations. We validated our thesis piecewise, by validating each application on its own. We used several validation techniques during this work. We first used visualization as it is a good way to familiarize oneself with the data. We then worked on program transformations and undertook a feasibility study to assess if our model was expressive enough to support these. Finally, we took a more empirical approach by defining benchmarks to measure and improve specific tasks.

We think that this strategy allowed us to gain a better understanding of the data, by first considering it at a high level before diving into details, in order to finally be able to isolate variables in a measurable manner.

10.4 Future Work

We have only scratched the surface of what can be done with accurate evolutionary data of systems. We plan to follow the following lines of work:

Gathering more data. Our data set is still restricted. To have more confidence in our results we need to replicate them on a wider sample of projects. We will record the development histories of more programmers in both the Smalltalk and Java versions of our systems. It would be ideal to have a public-access repository supporting automated application of benchmarks to new case studies.

Expanding the model with navigation and defect data. We have recorded additional IDE usage data such as navigation and execution data for an extended period of time, but have not used it yet. We envision enriching our model with this data and determine if this supports evolutionary tasks better.

Change documentation and task reconstruction. So far we have made the assumption that one development session is a development task. This is of course only an approximation: Some tasks span multiple sessions, while some sessions are made of several tasks. Tasks are also hierarchical and composed of subtasks. We plan to investigate how to automatically or semi-automatically split a development history in several distinct tasks.

Related to this, we plan to allow a developer to annotate changes for documentation purposes. During this process a developer could also specify the tasks in the development history he is documenting.

Clone detection and evolution. We want to use our data for the problem of clone detection and test whether it allows to detect clones more efficiently. Once this is done, we want to automatize the co-evolution of clones based on the program transformations that are at the moment done manually by the developer. Ideally we would define a benchmark for this task.

Example-based aspects. Continuing our work on program transformations, we want to deepen the idea of informal aspects we shown in Chapter 7, in particular to ease their transitions into more formal aspects, and to analyze their relationship with Hon and Kiczales's Fluid AOP [HK06].

Continuous reverse engineering. During the last two ICSE "Future Of Software Engineering" tracks, invited papers on the future of reverse engineering both mentioned continuous reverse engineering, *i.e.,* merging forward and reverse engineering in one smooth process. Our change-recording approach provides the immediacy (changes are gathered as they happen) and the interactivity (all tools are integrated in an IDE) necessary to fulfill such a vision. One first step towards this would be to devise an "Evolution Dashboard", allowing easy access to quick evolutionary facts about an entity as it is browsed.

Instant awareness. Several awareness systems exist, most working at the level of SCM transactions. With our approach, one can broadcast changes to all the developers of a system as soon as they are performed. Each development site could then check that the changes performed by other user do not break the code base. This could be done when merging changes, looking for conflicts while taking into account the semantics of the language, or at a higher level, checking if changes done by another developer break the tests of the code. If a conflict is found, a notification would be sent to conflicting parties so that they work together to fix the problem. In some cases, if the problem is repeatable, a transformation could be devised and stored to semi-automatically fix the following occurrences of the conflicts.

Repository-level conflict monitoring. Taking the previous idea to the next level, we envision a source code repository performing the same conflict checks, either at coding time or at checkin time if commits also include a change list. Conflicts would be checked automatically when libraries used by a project are changed. The author of a change would be informed of how many users have their code broken by it, while users of a library could easily see what they need to fix when upgrading to the next version of the library.

10.5 Closing Words

On impact. This work started with a radical idea which put us somewhat at odds with the software evolution community. It was hence recomforting to see that several approaches arrived to the same conclusions as ours, and even more so to see some of those were directly influenced by our approach.

In the field of model-driven engineering, Blanc *et al.* have shown that a change-based representation of models (without taking their history into account) allows for efficient checking of model validity [BMMM08]. Sriplakich *et al.* [SBG08] implemented a concept of *update commands* for collaborative model editing. Finally, Kögel described a change-based SCM system for models [KÖ8].

Fluri recognizes the need for a more detailed view of software evolution, and does so at by recovering fine-grained changes from SCM archives [FWPG07].

Several approaches are now recording changes in the IDE. Chan *et al.* [CCB07] record source code changes in Eclipse in a language-independent manner. Zumkher [Zum07] and Ebraert *et al.* [EVC$^+$07] record first-class changes in Smalltalk IDEs to provide dynamic software evolution, a scenario we have not tackled. Ebraert uses our model of first-class changes as a base for their change model. Lastly, Omori and Maruyama record Java-specific changes in Eclipse [OM08], and acknowledge our work as a direct inspiration.

Is it all worth it? The objective of this dissertation was to show that recording first-class changes is worthwhile. We think that the advantages (recording an accurate evolution with a model versatile enough to support several evolutionary tasks), outweighs the drawbacks (an approach requiring a lot of tool support and discarding data already available). Our results are significant improvements in several distinct areas that justify a further implementation of our approach.

Several researchers are working on the same or related problems. This makes us confident that the problem we address and its solution are on the right track.

Several factors have to be considered to make the approach practical:

- During this work we made sure the recording was as non-intrusive as possible. Tools soliciting the attention of developers when they want to focus on something else are problematic. This tradition needs to be continued.
- To further lower the barrier to entry, we need to fully integrate our approach with a standard versioning system. The changes we record can be exported in a simple text format, which could be simply put under version control.
- So far, we steered clear of optimizations. Experiments must be made to evaluate the requirements for large systems, and what are the costlier operations.

We are strongly convinced that Change-based Software Evolution provides a model of software integrating development and evolution tasks in a harmonious process. In the process we envision, reverse engineering and program comprehension tools are available in the IDE, merely a click away. The entire history is accessible, shared between developers, and contains all kinds of changes, from straightforward development and bug fixes to aspects and program transformations. Individual and shared recommenders continuously analyze the incoming stream of changes to extract valuable information and connect people.

Part V

Appendix

Appendix A

Inside the Change-based Repository

During the four yours we took to conclude this work, we recorded the history of 12 case studies, with large variations in size and style. We list them for reference, with some of their high-level characteristics, in Table A.1.

Project Name	Days	Classes	Methods	Lines of Code	Sessions	Developer Actions	Atomic Changes
SpyWare	1,095	697	7,243	37,347	496	23,227	548,384
Project X	98	498	2,280	14,109	121	5,513	141,580
Software Animator	62	605	1,682	8,418	133	15,723	108,940
Project A	7	17	228	687	17	903	23,512
Project B	7	35	340	1,450	19	1,595	38,168
Project C	7	41	357	1,537	22	1,326	38,818
Project D	8	20	260	1,931	19	757	26,789
Project E	7	16	117	760	10	311	8,672
Project F	8	15	142	1,191	17	511	19,522
Project G	6	23	197	580	10	842	12,851
Project H	7	10	159	1,347	22	597	18,237
Project I	7	50	454	1,952	22	1,326	25,965
Total	1,319	2,027	13,459	71,309	908	52,631	1,011,168

Table A.1: The case studies in our change repository

SpyWare is the research prototype we built to implement our vision of Change-based Software Evolution. SpyWare has been self-monitoring for more than three years. Included in the count are all the tools that we built to validate all approaches we presented, and the core of the platform itself. In this thesis, we used SpyWare's history in chapters 5, 6, 7, 8 and 9.

Project X contains the change history of a smalltalk developer working on a web application for a duration of three months. Of interest, the application was built using a preexisting web framework, implying a different usage pattern from other applications. We used project X's history in chapters 5 and 9.

Software Animator is a summer student project, developed over a duration of two months. It visualizes the change-based history of software systems in the form of a 3D animation, which can be played in real-time, or faster. It was implemented in Java, and monitored by EclipseEye, the Java version of our change recording framework. Software Animator was used in chapters 6 and 9.

Projects A to I are several student projects built in the course of a week. The students had the choice between three subjects, a virtual store (smaller projects), a geometry program (intermediate size) and a text-based role-playing game (larger projects). All projects were used in chapter 4, but only project I was reported on in detail. We also used the six largest of the nine projects (projects A, B, D, F, H and I) in our benchmarks in chapters 8 and 9, renumbering them A to F for clarity.

Bibliography

[ABF04] Erik Arisholm, Lionel C. Briand, and Audun Føyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[ABM$^+$05] Robert L. Akers, Ira D. Baxter, Michael Mehlich, Brian J. Ellis, and Kenn R. Luecke. Reengineering c++ component models via automatic program transformation. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[AGDS07] Erik Arisholm, Hans Gallis, Tore Dybå, and Dag I. K. Sjøberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[AK07] Eytan Adar and Miryung Kim. Softguess: Visualization and exploration of code clones in context. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 762–766, Washington, DC, USA, 2007. IEEE Computer Society.

[BAY03] James M. Bieman, Anneliese A. Andrews, and Helen J. Yang. Understanding change-proneness in oo software through visualization. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 44, Washington, DC, USA, 2003. IEEE Computer Society.

[BCH$^+$05] Dave Binkley, Mariano Ceccato, Mark Harman, Filippo Ricca, and Paolo Tonella. Automated refactoring of object oriented code into aspects. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 27–36, Washington, DC, USA, 2005. IEEE Computer Society.

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[BEJWKG05] Jennifer Bevan, Jr. E. James Whitehead, Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. *SIGSOFT Software Engineering Notes*, 30(5):177–186, 2005.

[BGD$^+$06] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143, New York, NY, USA, 2006. ACM.

[BGH07] Marat Boshernitsan, Susan L. Graham, and Marti A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 567–576, New York, NY, USA, 2007. ACM.

[BMMM08] Xavier Blanc, Isabelle Mounier, Alix Mougenot, and Tom Mens. Detecting model inconsistency through operation-based model construction. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 511–520, New York, NY, USA, 2008. ACM.

[Bra94] Stewart Brand. *How Buildings Learn - What Happens After They're Built*. Penguin Books, 1994.

[BvDT05] Magiel Bruntink, Arie van Deursen, and Tom Tourwe. Isolating idiomatic crosscutting concerns. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 37–46, Washington, DC, USA, 2005. IEEE Computer Society.

[BWL99] Lionel C. Briand, Juergen Wuest, and Hakim Lounis. Using coupling measurement for impact analysis in object-oriented systems. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 475, Washington, DC, USA, 1999. IEEE Computer Society.

[CCB07] Jacky Chan, Alan Chu, and Elisa Baniassad. Supporting empirical studies by non-intrusive collection and visualization of fine-grained revision history. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 60–64, New York, NY, USA, 2007. ACM.

[CCP07] Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. Identifying changed source code lines from version repositories. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[CMR04] Andrea Capiluppi, Maurizio Morisio, and Juan F. Ramil. The evolution of source folder structure in actively evolved open source systems. In *METRICS '04: Proceedings of the Software Metrics, 10th International Symposium*, pages 2–13, Washington, DC, USA, 2004. IEEE Computer Society.

[Cor89] Thomas A. Corbi. Program understanding: challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[CP07] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. In *FOSE '07: Proceedings of the 2nd Conference on the Future of Software Engineering*, pages 326–341, Washington, DC, USA, 2007. IEEE Computer Society.

[CSY$^+$04] Kai Chen, Stephen R. Schach, Liguo Yu, Jeff Offutt, and Gillian Z. Heller. Open-source change logs. *Empirical Software Engineering*, 9(3):197–210, 2004.

[CW98] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[DB06] Saeed Dehnadi and Richard Bornat. The camel has two humps (working title). 2006.

[DCMJ06] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. Automated detection of refactorings in evolving components. In Springer, editor, *ECOOP '06:ECOOP '06: Proceedings of the 20th European Conference on Object Oriented Programming*, Lecture Notes in Computer Science, pages 404–428, 2006.

[DCR05] Robert DeLine, Mary Czerwinski, and George Robertson. Easing program comprehension by sharing navigation data. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 241–248, Washington, DC, USA, 2005. IEEE Computer Society.

[DDN00] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 166–177, New York, NY, USA, 2000. ACM.

[DDT99] Serge Demeyer, Stéphane Ducasse, and Sander Tichelaar. Why unified is not universal. uml shortcomings for coping with round-trip engineering. In *UML '99: Proceedings of The Second International Conference on The Unified Modeling Language*, Lecture Notes in Computer Science, pages 630–645. Springer, 1999.

[DER07] Ekwa Duala-Ekoko and Martin P. Robillard. Tracking code clones in evolving software. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 158–167, Washington, DC, USA, 2007. IEEE Computer Society.

[DGL$^+$07] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli, and Pascal Zumkehr. Encapsulating and exploiting change with changeboxes. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 25–49, New York, NY, USA, 2007. ACM.

[DJ05] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[DL06a] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 189–198, Washington, DC, USA, 2006. IEEE Computer Society.

[DL06b] Marco D'Ambros and Michele Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 229–238, Washington, DC, USA, 2006. IEEE Computer Society.

[DLG05] Marco D'Ambros, Michele Lanza, and Harald Gall. Fractal figures: Visualizing development effort for cvs entities. In *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 16, Washington, DC, USA, 2005. IEEE Computer Society.

[DMJN07] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Computer Society.

[DNMJ08] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. Reba: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 441–450, New York, NY, USA, 2008. ACM.

[DR08] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 481–490, New York, NY, USA, 2008. ACM.

[EA04] Torbjörn Ekman and Ulf Asklund. Refactoring-aware versioning in eclipse. *Electronic Notes in Theoritical Computer Science*, 107:57–69, 2004.

[EG05] Jacky Estublier and Sergio Garcia. Process model and awareness in scm. In *SCM '05: Proceedings of the 12th international workshop on Software configuration management*, pages 59–74, New York, NY, USA, 2005. ACM.

[EG06] Jacky Estublier and Sergio Garcia. Concurrent engineering support in software engineering. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 209–220, Washington, DC, USA, 2006. IEEE Computer Society.

[EGK$^+$01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[ELvdH$^+$05] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy 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, 2005.

[Erl00] Len Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[ES98] Katalin Erdös and Harry M. Sneed. Partial comprehension of complex programs (enough to perform maintenance). In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 98, Washington, DC, USA, 1998. IEEE Computer Society.

[Est95] Jacky Estublier. *The Adele configuration manager*, pages 99–133. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[EVC$^+$07] Peter Ebraert, Jorge Vallejos, Pascal Costanza, Ellen Van Paesschen, and Theo D'Hondt. Change-oriented software engineering. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 3–24, New York, NY, USA, 2007. ACM.

[EZS$^+$08] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008.

[FG06] Beat Fluri and Harald C. Gall. Classifying change types for qualifying change couplings. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 35–45, Washington, DC, USA, 2006. IEEE Computer Society.

[Fow02] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Springer-Verlag, London, UK, 2002.

[FPG03] Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society.

[Fre07] Tammo Freese. Operation-based merging of development histories. In *WRT '07: Proceedings of the 1st ECOOP Workshop on Refactoring Tools*, 2007.

[FWPG07] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[GB80] Ira P. Goldstein and Daniel G. Bobrow. A layered approach to software design. Technical Report CSL-80-5, Xerox PARC, December 1980.

[GDL04] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 40–49, Washington, DC, USA, 2004. IEEE Computer Society.

[GHJ98] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[Gîr05] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[GJK03] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[GJKT97] Harald Gall, Mehdi Jazayeri, René Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *ICSM '97: Proceedings of the International Conference on Software Maintenance*, page 160, Washington, DC, USA, 1997. IEEE Computer Society.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 113–122, Washington, DC, USA, 2005. IEEE Computer Society.

[GKY91] Bjørn Gulla, Even-André Karlsson, and Dashing Yeh. Change-oriented version descriptions in epos. *Software Engineering Journal*, 6(6):378–386, 1991.

[GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *CSMR '05: Proceedings of the Ninth European Conference on Software Maintenance and Reengineering*, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[GT00] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *ICSM '00: Proceedings of the International Conference on Software Maintenance (ICSM'00)*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[GZ05] Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[HD05] Johannes Henkel and Amer Diwan. Catchup!: capturing and replaying refactorings to support api evolution. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 274–283, New York, NY, USA, 2005. ACM.

[HGBR08] Israel Herraiz, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Determinism and evolution. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 1–10, New York, NY, USA, 2008. ACM.

[HH06] Ahmed E. Hassan and Richard C. Holt. Replaying development history to assess the effectiveness of change propagation tools. *Empirical Software Engineering*, 11(3):335–367, 2006.

[HK06] Terry Hon and Gregor Kiczales. Fluid aop join point models. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 712–713, New York, NY, USA, 2006. ACM.

[HN86] A. Nico Habermann and David Notkin. Gandalf: software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, 1986.

[HP96] Richard C. Holt and J. Y. Pak. Gase: visualizing software evolution-in-the-large. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 163, Washington, DC, USA, 1996. IEEE Computer Society.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *MoDELS Satellite Events*, pages 128–138, 2005.

[K08] Maximilian Kögel. Towards software configuration management for unified models. In *CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 19–24, New York, NY, USA, 2008. ACM.

[Kag07] Huzefa Kagdi. Improving change prediction with fine-grained source code mining. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 559–562, New York, NY, USA, 2007. ACM.

[KG06] Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society.

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KM05] Mik Kersten and Gail C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM.

[KM06] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11, New York, NY, USA, 2006. ACM.

[KNG07] Miryung Kim, David Notkin, and Dan Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 333–343, Washington, DC, USA, 2007. IEEE Computer Society.

[KPEJW05] Sunghun Kim, Kai Pan, and Jr. E. James Whitehead. When functions change their names: Automatic detection of origin relationships. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 143–152, Washington, DC, USA, 2005. IEEE Computer Society.

[KSB08] Marouane Kessentini, Houari A. Sahraoui, and Mounir Boukadoum. Model transformation as an optimization problem. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 159–173. Springer, 2008.

[KSNM05] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Software Engineering Notes*, 30(5):187–196, 2005.

[LAEW08] Jonathan Lung, Jorge Aranda, Steve M. Easterbrook, and Gregory V. Wilson. On the difficulty of replicating human subjects studies in software engineering. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 191–200, New York, NY, USA, 2008. ACM.

[Lan01] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *IWPSE '01: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 37–42, New York, NY, USA, 2001. ACM.

[LB85] Meir M. Lehman and Laszlo A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Lip92] Ernst Lippe. *CAMERA – Support for distributed cooperative work*. PhD thesis, University of Utrecht, 1992.

[LL07] Mircea Lungu and Michele Lanza. Exploring inter-module relationships in evolving software systems. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 91–102, Washington, DC, USA, 2007. IEEE Computer Society.

[LLG06] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[LM05] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[LRW$^+$97] Meir M. Lehman, Juan F. Ramil, Paul Wernick, Dewayne E. Perry, and Wladyslaw M. Turski. Metrics and laws of software evolution - the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA, 1997. IEEE Computer Society.

[LvO92] Ernst Lippe and Norbert 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.

[LW07] Jacob Lehraum and Bill Weinberg. Ide evolution continues beyond eclipse. `http://www.eetimes.com/article/showArticle.jhtml?articleId=21400991`, 2007.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. *SIGPLAN Notices*, 22(12):147–155, 1987.

[Mar04] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[MFV$^+$05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel. On executable meta-languages applied to model transformations. Model Transformations In Practice Workshop, 2005.

[MJS$^+$00] Hausi A. Müller, Jens H. Jahnke, Dennis B. Smith, Margaret-Anne Storey, Scott R. Tilley, and Kenny Wong. Reverse engineering: a roadmap. In *FOSE '00: Proceedings of the 1st Conference on The future of Software engineering*, pages 47–60. ACM Press, 2000.

[MKF06] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[NMBT05] Tien N. Nguyen, Ethan V. Munson, John T. Boyland, and Cheng Thao. An infrastructure for development of object-oriented, multi-level configuration management services. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 215–224, New York, NY, USA, 2005. ACM.

[OM08] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 31–34, New York, NY, USA, 2008. ACM.

[Ost87] Leon J. Osterweil. Software processes are software too. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 2–13, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[PBMW98] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998.

[Per87] Dewayne E. Perry. Version control in the inscape environment. In *ICSE '87: Proceedings of the 9th international conference on Software Engineering*, pages 142–149, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[PG06] Chris Parnin and Carsten Gorg. Building usage contexts during program comprehension. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 13–22, Washington, DC, USA, 2006. IEEE Computer Society.

[PGM$^+$07] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[PGR06] Chris Parnin, Carsten Görg, and Spencer Rugaber. Enriching revision history with interactions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 155–158, New York, NY, USA, 2006. ACM.

[Pin99] Steven Pinker. *How the mind works*. Penguin Books, Harmondsworth, Middlessex (UK), 1999.

[Pin05] Martin Pinzger. *ArchView – Analyzing Evolutionary Aspects of Complex Software Systems*. PhD thesis, Vienna University of Technology, 2005.

[PM06] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 469–478, Washington, DC, USA, 2006. IEEE Computer Society.

[PP05] Ranjith Purushothaman and Dewayne E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[RAGBH05] Gregorio Robles, Juan Jose Amor, Jesus M. Gonzalez-Barahona, and Israel Herraiz. Evolution and growth in large libre software projects. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 165–174, Washington, DC, USA, 2005. IEEE Computer Society.

[RB04] Don Roberts and John Brant. Tools for making impossible changes - experiences with a tool for transforming large smalltalk programs. *IEE Proceedings - Software*, 152(2):49–56, April 2004.

[RDB$^+$07] Coen De Roover, Theo D'Hondt, Johan Brichau, Carlos Noguera, and Laurence Duchien. Behavioral similarity matching using concrete source code templates in logic queries. In *PEPM '07: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 92–101, New York, NY, USA, 2007. ACM.

[RDGM04] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *CSMR '04: Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, page 223, Washington, DC, USA, 2004. IEEE Computer Society.

[Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[RL05] Romain Robbes and Michele Lanza. Versioning systems for evolution research. In *IWPSE '05: Proceedings of the 8th International Workshop on Principles of Software Evolution*, pages 155–164. IEEE CS Press, 2005.

[RL06] Romain Robbes and Michele Lanza. Change-based software evolution. In *EVOL '06: Proceedings of the 2nd International ERCIM Workshop on Challenges in Software Evolution*, pages 159–164, 2006.

[RL07a] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, January 2007.

[RL07b] Romain Robbes and Michele Lanza. Characterizing and understanding development sessions. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 155–166, Washington, DC, USA, 2007. IEEE Computer Society.

[RL07c] Romain Robbes and Michele Lanza. Towards change-aware development tools. Technical Report 6, Faculty of Informatics, Università della Svizzerra Italiana, Lugano, Switzerland, may 2007.

[RL08a] Romain Robbes and Michele Lanza. Example-based program transformation. In *MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 174–188. Springer, 2008.

[RL08b] Romain Robbes and Michele Lanza. How program history can improve code completion. In *ASE 08: Proceedings of the 23rd ACM/IEEE International Conference on Automated Software Engineering*, pages 317–326. ACM Press, 2008.

[RL08c] Romain Robbes and Michele Lanza. Spyware: A change-aware development toolset. In *ICSE '08: Proceedings of the 30th International Conference in Software Engineering*, pages 847–850. ACM Press, 2008.

[RLL07] Romain Robbes, Michele Lanza, and Mircea Lungu. An approach to software evolution based on semantic change. In *FASE '07: Proceedings of 10th International Conference on the Fundamentals of Software Engineerings*, Lecture Notes in Computer Science, pages 27–41. Springer, 2007.

[RLP08] Romain Robbes, Michele Lanza, and Damien Pollet. A benchmark for change prediction. Technical Report 06, Faculty of Informatics, Università della Svizzerra Italiana, Lugano, Switzerland, October 2008.

[Rob07] Romain Robbes. Mining a change-based software repository. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 15, Washington, DC, USA, 2007. IEEE Computer Society.

[Roc75] Marc J. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

[RPL08] Romain Robbes, Damien Pollet, and Michele Lanza. Logical coupling based on fine-grained change information. In *WCRE '08: Proceedings of the 15th IEEE International Working Conference on Reverse Engineering*, pages 42–46. IEEE Press, 2008.

[SBG08] Prawee Sriplakich, Xavier Blanc, and Marie-Pierre Gervals. Collaborative software engineering on large-scale models: requirements and experience in modelbus. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, pages 674–681, New York, NY, USA, 2008. ACM.

[SEH03] Susan Elliott Sim, Steve Easterbrook, and Richard C. Holt. Using benchmarking to advance research: a challenge to software engineering. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 74–83, Washington, DC, USA, 2003. IEEE Computer Society.

[SES05] Janice Singer, Robert Elves, and Margaret-Anne Storey. Navtracks: Supporting navigation in software maintenance. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 325–334, Washington, DC, USA, 2005. IEEE Computer Society.

[SGPP04] Kevin A. Schneider, Carl Gutwin, Reagan Penner, and David Paquette. Mining a software developer's local interaction history. In *MSR '04: Proceedings of the 1st international workshop on Mining Software Repositories*, pages 106–110, Los Alamitos CA, 2004. IEEE Computer Society Press.

[SH01] Till Schümmer and Jörg M. Haake. Supporting distributed software development by modes of collaboration. In *ECSCW'01: Proceedings of the seventh conference on European Conference on Computer Supported Cooperative Work*, pages 79–98, Norwell, MA, USA, 2001. Kluwer Academic Publishers.

[Sha07] Yuval Sharon. Eclipseye - spying on eclipse. Bachelor's thesis, University of Lugano, June 2007.

[SHE02] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate c++ extractors. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.

[SLM03] Jelber Sayyad Shirabad, Timothy C. Lethbridge, and Stan Matwin. Mining the maintenance history of a legacy software system. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 95, Washington, DC, USA, 2003. IEEE Computer Society.

[Smi84] Brian Cantwell Smith. Reflection and semantics in lisp. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35, New York, NY, USA, 1984. ACM.

[SNvdH03] Anita Sarma, Zahra Noroozi, and André van der Hoek. Palantír: raising awareness among configuration management workspaces. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 444–454, Washington, DC, USA, 2003. IEEE Computer Society.

[Som06] Ian Sommerville. *Software Engineering: (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[SRG08] Ilie Savga, Michael Rudolf, and Sebastian Goetz. Comeback!: a refactoring-based tool for binary-compatible framework upgrade. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 941–942, New York, NY, USA, 2008. ACM.

[TBG04] Michael Toomim, Andrew Begel, and Susan L. Graham. Managing duplicated code with linked editing. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, pages 173–180, Washington, DC, USA, 2004. IEEE Computer Society.

[TCS05] Nikolaos Tsantalis, Alexander Chatzigeorgiou, and George Stephanides. Predicting the probability of change in object-oriented systems. *IEEE Transactions on Software Engineering*, 31(7):601–614, 2005.

[TDD00] Sander Tichelaar, Stéphane Ducasse, and Serge Demeyer. Famix and xmi. In *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, page 296, Washington, DC, USA, 2000. IEEE Computer Society.

[TDX07] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of api refactorings in libraries. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 377–380, New York, NY, USA, 2007. ACM.

[TG02] Qiang Tu and Michael W. Godfrey. An integrated approach for studying architectural evolution. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 127, Washington, DC, USA, 2002. IEEE Computer Society.

[Tic85] Walter F. Tichy. Rcs—a system for version control. *Software Practice and Experience*, 15(7):637–654, 1985.

[TM02] Christopher M. B. Taylor and Malcolm Munro. Revision towers. In *VISSOFT '02: Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, page 43, Washington, DC, USA, 2002. IEEE Computer Society.

[Tuf06] Edward R. Tufte. *Beautiful Evidence*. Graphis Pr, 2006.

[Var06] Dániel Varró. Model transformation by example. In *Models '06: Proceedings of the 9th international conference on Model Driven Engineering Languages and Systems*, Lecture Notes in Computer Science, pages 410–424. Springer, 2006.

[Vis02] Eelco Visser. Meta-programming with concrete object syntax. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*, pages 299–315, London, UK, 2002. Springer-Verlag.

[vR79] C. J. van Rijsbergen. *Information Retrieval*. Butterworth, 2nd edition edition, 1979.

[WD06] Peter Weissgerber and Stephan Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[WHH04] Jingwei Wu, Richard C. Holt, and Ahmed E. Hassan. Exploring software evolution using spectrographs. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 80–89, Washington, DC, USA, 2004. IEEE Computer Society.

[WK00] F. George Wilkie and Barbara A. Kitchenham. Coupling measures and change ripples in c++ application software. *Journal of Systems and Software*, 52(2-3):157–164, 2000.

[WL08] Richard Wettel and Michele Lanza. Visual exploration of large-scale system evolution. In *WCRE '08: Proceedings of the 15th IEEE International Working Conference on Reverse Engineering*, pages 219–228. IEEE CS Press, 2008.

[WSKK07] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards model transformation generation by-example. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 285b, Washington, DC, USA, 2007. IEEE Computer Society.

[Wuy07] Roel Wuyts. Roeltyper. `http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/`, 2007.

[XS05] Zhenchang Xing and Eleni Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *IEEE Transactions on Software Engineering*, 31(10):850–868, 2005.

[YMNCC04] Annie T. T. Ying, Gail C. Murphy, Raymond T. Ng, and Mark Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions Software Engineering*, 30(9):574–586, 2004.

[Zel07] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *FOSE '07: Proceedings of the 2nd Conference on The Future of Software Engineering*, pages 316–325, Washington, DC, USA, 2007. IEEE Computer Society.

[ZGH07] Lijie Zou, Michael W. Godfrey, and Ahmed E. Hassan. Detecting interaction coupling from task interaction histories. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 135–144, Washington, DC, USA, 2007. IEEE Computer Society.

[ZS95] Andreas Zeller and Gregor Snelting. Handling version sets through feature logic. In *Proceedings of the 5th European Software Engineering Conference*, pages 191–204, London, UK, 1995. Springer-Verlag.

[Zum07] Pascal Zumkehr. Changeboxes — modeling change as a first-class entity. Master's thesis, University of Bern, February 2007.

[ZWDZ04] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.