TOWARDS A CHANGE-ORIENTED PROGRAMMING ENVIRONMENT

BY

STANISLAV NEGARA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph Johnson, Chair
Associate Professor Darko Marinov
Associate Professor Samuel Kamin
Visiting Research Assistant Professor Daniel Dig
Professor Andrew Black, Portland State University

Abstract

Any successful software system continuously evolves in response to ever-changing requirements. Developers regularly add new or adjust existing features, fix bugs, tune performance, etc. Thus, *code change* is the heart of software development.

Nevertheless, the traditional programming environments and toolkits treat changes as a by-product of software evolution. First, most of the tools do not offer direct access to the history of code changes and do not allow to browse or manipulate these changes. Second, different tools represent changes differently and thus, do not understand each other. For example, text editors treat changes to the edited document at the character level, Version Control Systems (VCS) track changes as text line differences in the corresponding file, and refactoring engines represent changes with high-level descriptors. Third, the most common, textual representation of changes is inappropriate for a developer, who reasons in terms of program entities rather than individual characters or text lines. Hence, the current tools' support for the major activity of a developer, code changing, is limited.

To address these limitations, we propose to make change the first-class citizen and place it at the center of a programming environment. In our approach, we represent any change, from low-level code edits up to high-level automated refactorings, as a *program transformation*. Consequently, we express a version of a program as a sequence of such program transformations.

We developed CodingTracker, an infrastructure that captures all changes that affect a program's code. CodingTracker uniformly represents the raw captured changes as operations on the corresponding Abstract Syntax Tree (AST) nodes. These AST node operations are the lowest level of program transformations that serve as building blocks for higher level transformations. Also, CodingTracker implements a replayer that enables a developer to browse and replay the captured code changes. We employed CodingTracker to perform a field study and answered several important code evolution research questions that could not be answered without such infrastructure.

As part of CodingTracker, we implemented an algorithm that infers refactorings from the AST node operations. Refactorings are high-level program transformations that improve the underlying code's design

without affecting its behavior. Our algorithm infers refactorings from *continuous* code changes, and thus, it could be applied on-line, assisting a developer with the refactoring process. Moreover, our refactoring inference algorithm enabled us to perform a field study that compared the practice of manual and automated refactorings. Our findings reveal new facts about the refactoring practice that benefit code evolution researchers, tool builders, and developers.

Finally, we augmented CodingTracker with an algorithm that helps identify high-level program transformations without prior knowledge of how these transformations look. Our novel algorithm employs data mining techniques to detect frequent code change patterns. We applied our algorithm on the data collected in our field study and showed that it is scalable and effective. Analyzing the detected code change patterns, we identified ten kinds of popular program transformations. Our results shed light on how developers evolve their code as well as enable tool builders to automate common program transformations.

To my parents.

Acknowledgments

I would like to thank my adviser, Ralph Johnson, for his valuable support and guidance throughout my PhD Program; Danny Dig for numerous inspirational discussions on making change the first-class citizen; Darko Marinov, Sam Kamin, and Andrew Black for their insightful comments on the earlier drafts of my dissertation.

I am thankful to Mohsen Vakilian and Nick Chen for the collaboration on conducting the field study; Mihai Codoban for his help with identifying high-level program transformations out of the mined frequent code change patterns; Nathan Hirtz for his suggestions on statistical sampling techniques; and Jiawei Han for his guidance in the field of data mining.

I am grateful to Cosmin Radoi, Milos Gligoric, Samira Tasharofi, Semih Okur, Jeff Overbey, Caius Brindescu, Yu Lin, Yun Young Lee, Vilas Jagannath, Qingzhou Luo, Adrian Nistor, Kyle Doren, Lorand Szacaks, Alex Gyori, Lyle Franklin, and anonymous reviewers for their feedback on the drafts of the papers that contributed to this dissertation.

During the work on my dissertation, I was partially supported by the Institute for Advanced Computing Applications and Technologies (IACAT) at the University of Illinois at Urbana-Champaign, by the United States Department of Energy under Contract No. DE-F02-06ER25752, and by the National Science Foundation award number CCF 11-17960.

Table of Contents

Chapter 1

Introduction

1.1 Change is the Heart of Software Development

Any successful software system continuously evolves in response to ever-changing requirements [88]. Developers spend most of their time changing the existing software systems, while companies spend more on maintaining old software systems than on developing new ones. In particular, it is a known fact that at least two-thirds of software costs are due to evolution [10, 16, 24, 42, 57, 91, 146], with some industrial surveys [40] claiming 90%.

There are many kinds of changes that developers regularly perform on their code. For example, developers add new features to expand the capabilities of a software system, fix bugs in the existing features, and perform refactorings to improve the system's design without affecting its behavior. Also, it is common that one change leads to other changes in the evolving code. In particular, implementing a new feature requires adding new tests and/or updating some of the existing tests. Fixing a bug in an already implemented feature is another frequent activity of developers that causes a similar ripple effect on the tests. Moreover, developers change a software system in response to changes that are external to it. For example, changes in the underlying hardware might require re-optimizing the system's code. Similarly, changes in the used libraries might require adapting the system to the updated library APIs.

In spite of the pervasive nature of code changes, the modern development toolkits provide inadequate support for them. Consequently, developers predominantly perform changes as manual text edits (refactorings automated in some modern IDEs are the rare exceptions). Manual text edits are not only tedious and error-prone, but also too low-level for browsing and understanding changes retrospectively. As a result, such changes are never reused.

Code evolution researchers also face the problem of retrospective change understanding. Since low-level textual changes are not suitable for such tasks, researchers [5, 43, 52, 78, 79, 82, 134, 139] designed tools that infer structural high-level changes in the evolving code. Unfortunately, the primary source of data for this inference are Version Control System (VCS) snapshots, which lack a significant fraction of code evolution

information [104].

Our vision is that code change should be placed at the center of a programming environment. In a change-oriented programming environment any change should be treated as a program transformation. Thus, programmers would develop their code as a continuous application of program transformations, which could be customized, combined, and easily accessed for understanding, reusing, adjusting, undoing, etc. Our environment would enable discovery and automation of popular program transformations. However, we envision that automating all program transformations that a developer might need is an infeasible task. Therefore, the proposed environment should simplify crafting of custom program transformations that a developer would create once and reuse as many times as needed. Also, a change-oriented programming environment should provide a collection of basic program transformations and support their composition such that developers could easily construct more high-level program transformations out of the lower-level ones.

In the following, we present the current problems of code evolution researchers and developers in more detail and show how we address some of these problems.

1.2 Traditional Data Sources for Software Evolution Research are Limited

Software evolution research extracts the code evolution information from the historical data of an application. The traditional source of this historical data is a file-based Version Control System (VCS). File-based VCSs are very popular among developers (e.g., Git [51], SVN [127], CVS [27]). Therefore, software evolution researchers [1,31,32,39,41,46–48,50,53,61,64,71,73,83,89,117,125,126,135,137,138,149] use VCS to access the historical data of many software systems. Although convenient, using VCS code evolution data for software evolution research is inadequate.

First, it is *incomplete*. A single VCS commit may contain hours or even days of code development. During this period, a developer may change the same code fragment multiple times, for example, tuning its performance, or fixing a bug. Therefore, there is a chance that a subsequent code change would override an earlier change, thus *shadowing* it. Since a shadowed change is not present in the code, it is not present in the snapshot committed to VCS. Therefore, code evolution research performed on the snapshots stored in the VCS (like [46–48]) does not account for shadowed code changes. Ignoring shadowed changes could significantly limit the accuracy of tools that try to infer the intent of code changes (e.g., infer refactorings [31, 32,53,135,137], infer bug fixes [62,81,84,86,116,148]). For example, if a method is renamed more than once,

a snapshot-based analysis would infer only the last refactoring.

Second, VCS data is *imprecise*. A single VCS commit may contain several overlapping changes to the same program entity. For example, many times refactorings overlap with editing sessions, e.g., a method is both renamed, and its method body is changed dramatically. Refactorings can also overlap with other refactorings, e.g., a method is both renamed and its arguments are reordered. This overlap makes it harder to infer the intent of code changes.

Third, answering research questions that correlate code changes with other development activities (e.g., test runs, automated refactorings) is *impossible*. VCS is limited to code changes, and does not capture many kinds of other developer actions: running the application or the tests, invoking automated refactorings from the IDE, etc. This severely limits the ability to study the code development process. What is the proportion of manual vs. automated refactorings? How often do developers commit changes that are untested? How often do they fix assertions in the failing tests rather then fixing the system under test? These are all examples of questions that cannot be answered using VCS data only.

1.3 Code Developers Lack Tools for Effective Change Management

Although *code change* is the heart of software development, the conventional programming environments and toolkits provide inadequate support for managing change.

Common code development tools do not allow a developer to access the history of code changes. For example, if a developer fixes a bug using a text editor, he sees only the resulting code rather than a sequence of changes that transformed the original, buggy code into the correct one. A developer might be interested in the changes that fix a bug for a variety of reasons. For example, similar changes might have to be applied to an instance of the same bug in a different part of the code. Or, the developer could report this sequence of changes to facilitate the fix review.

Eclipse [37] refactoring history is a rare example when a developer can access the sequence of a specific kind of high-level changes, refactorings, that were automatically applied to the developed application. Unfortunately, this history log is static and does not allow a developer to manipulate the contained refactorings. A developer might be interested in manipulating high-level changes in order to adjust them for a specific task. For example, while optimizing a loop, in addition to browsing the optimizations applied to this loop, a developer might need to manipulate some of them to achieve better performance, e.g., undo those that are redundant or adjust such parameters as tile size or unroll factor.

Another problem is that different tools represent changes differently. In particular, text editors treat changes to the edited document at the character level, Version Control Systems (VCS) track changes as text line differences in the corresponding file, and refactoring engines represent changes with high-level descriptors. Consequently, different tools do not cooperate well. For example, traditional VCS do not understand that a particular line is changed as a result of a refactoring. Consider the scenario in which a single code line represents an expression involving a variable. Two developers check out the code containing this line. One developer renames this variable, while another developer changes the expression's operation. Although these changes affect the same line, they are not conflicting and thus, should be merged without problems. Nevertheless, if these two developers commit their changes to the repository (in any order), VCS will report a spurious syntactic conflict to the second committer.

Arguably the most important problem is that the vast majority of tools represent changes in textual form. Textual representation is too low-level and irregular. It does not express well the intent of a developer, who reasons in terms of program entities rather than individual characters or text lines. For example, a single textual change could represent typing a single character or inserting a whole class declaration. Moreover, even if several textual changes are equivalent in the number of affected characters, they could have a totally different impact on the underlying code depending on whether they represent editing a comment, typing a long name of a single variable, or adding several short statements.

Another limitation of textual representation is that it hinders expression of high-level changes as a composition of multiple low-level changes. Such expression is crucial to enable cooperation of tools that operate on different levels, e.g., text editors and automated refactoring engines. Besides, flexible composition of several changes will both allow a developer to create new kinds of changes and enable a programming environment to effectively automate them.

Although many code changes are repetitive by nature, and thus form code change *patterns*, the existing IDEs cannot identify such changes *on-the-fly*, i.e., when the changes are in progress. Identifying repetitive changes on-the-fly would enable an IDE to better support them, e.g., by performing such changes automatically or checking their consistency. The existing research [17, 20, 21, 66, 90, 96, 128, 130, 141, 143, 150] predominantly detects frequent code change patterns either analyzing the static source code of an application or comparing the application's VCS snapshots, neither of which is suitable to learn code changes as soon as they happen. Recent research [44, 49] aimed at providing on-the-fly code change assistance to developers, but their code change identification techniques were limited in two ways: (i) they were looking for a single kind of code change pattern — refactorings, (ii) they considered only a small subset of previously known kinds of refactorings.

1.4 Thesis Statement

Our goal is to address some of the limitations of current programming environments mentioned above by moving change to a higher level of abstraction, making it first-class citizen, and placing it at the center of a programming environment. Our *thesis* is that *a programming environment that focuses on code changes rather than the code itself is implementable and has the following advantages over a traditional environment:*

- *It allows code evolution researchers to capture more fine-grained and precise changes to the code.*
- *It enables researchers to infer refactorings from manual code edits in real time.*
- *It makes it possible for researchers to automate the detection of frequent code change patterns that help identify previously unknown high-level program transformations that developers repeatedly apply to different code fragments.*

Although in this dissertation we show the usefulness of placing code change at the center of a programming environment for researchers only, we think that our approach and the developed infrastructure is the first step to address the problems of developers as well. In particular, we have not formally evaluated the usefulness of our record/replay infrastructure, but we think that developers would benefit from it. For example, developers can employ this infrastructure to access the detailed history of code changes and thus, improve their understanding of code evolution. Moreover, our infrastructure could be augmented with more advanced UIs that will facilitate such understanding (e.g., by grouping changes along such dimensions as time, a specific program entity, a particular developer, etc.). Additionally, our refactoring inference algorithm can be applied on-line and thus, serve as the core of a tool that offers continuous support of a developer's refactoring activity. Such a tool could automatically complete a refactoring that a developer started to perform manually or check correctness of a refactoring that the developer completed manually. Finally, the results of our empirical studies can be used to guide the work of tool builders, e.g., they can prioritize automation of those refactoring kinds that are popular among developers.

1.5 A Change-Oriented Programming Environment

1.5.1 Capturing and Representing Detailed Code Evolution Data

To overcome the limitations of the traditional data sources of software evolution researchers, we track all changes that directly or indirectly affect the underlying code. We developed a tool, CodingTracker, an Eclipse plug-in that unintrusively collects fine-grained data about code evolution of Java programs. In particular,

CodingTracker records every code edit performed by a developer. It also records many other developer actions, for example, invocations of automated refactorings, tests and application runs, interactions with Version Control System (VCS), etc. The collected data is so precise that it enables us to reproduce the state of the underlying code at any point in time.

Individual code edits collected by CodingTracker represent code-changing actions of a developer in the most precise way, but they are too irregular to serve as a unit of change in a code evolution analysis. Therefore, we define a unit of code change as an atomic operation on an Abstract Syntax Tree (AST) node: *add*, *delete*, or *update*, where *add* adds a node to AST, *delete* removes a node from AST, and *update* changes a property of an existing AST node (e.g., name of a variable). Note that *move* operation, which moves a child node from one parent to another one in an AST, is not atomic, but rather consists of two consequent AST node operations: *delete* and *add*. Thus, we do not consider *move* a unit of code change.

In contrast to raw code edits, AST node operations are a higher level of abstraction and can serve as building blocks for high-level transformations in the proposed programming environment. For example, a refactoring could be expressed either with a high-level descriptor or with a sequence of AST node operations (as well as some other basic operations like deleting or moving a source code file) that represent the refactoring's effects on the underlying code.

We developed a novel algorithm [104] that infers AST node operations from the collected raw code edits. Our algorithm takes as input a sequence of code changes, infers the corresponding AST node operations, and inserts the inferred operations in the original sequence such that every AST node operation immediately follows the code edits from which it was inferred.

To demonstrate the benefits of capturing more detailed data about code evolution as well as quantify the limitations of VCS snapshots, we performed a field study on 24 developers working in their natural settings. We asked our participants to install CodingTracker in their Eclipse IDEs. During the study, CodingTracker collected data for 1,652 hours of development, which involve 2,000 commits comprising 23,002 committed files, and 9,639 test session runs involving 314,085 testcase runs.

The collected data enabled us to answer five research questions that could not be answered without a tool like CodingTracker. In particular, we found that 37% of code changes are *shadowed* by other changes, and are not stored in VCS. Also, developers intersperse different kinds of changes in the same commit. For example, 46% of refactored program entities are also edited in the same commit. Both these findings provide quantitative evidence of how incomplete and imprecise VCS data is.

1.5.2 Accessing and Replaying the History of Code Changes

As part of CodingTracker, we implemented a replayer that enables a developer to directly access the full history of code changes. Our replayer allows a developer to load a recorded sequence of code changes, browse it, hide kinds of changes that the developer is not interested in (e.g., invocations of automated refactorings or textual edits), and replay the sequence at any desired pace. CodingTracker's replayer is an Eclipse View that is displayed alongside other Views of an Eclipse workbench, thus enabling the developer to see the results of the replayed changes in the same Eclipse instance. The developer can use breakpoints to stop the replaying at specific points. Also, the developer can reset the replaying and start over as many times as needed.

The developer can select any code change in the list and see its detailed information in the bottom pane of the replayer. For a simple code edit, this information includes the new and the old text, the offset, and the length of the deleted text. For an invocation of automated refactoring, this information includes the refactoring kind, the handler of the refactored program entity, the configuration options of the refactoring, etc. Also, for all code changes, the replayer displays the timestamp at which the corresponding change took place.

1.5.3 Inferring Refactorings from Continuous Code Changes

Refactoring [45] is an important part of software development. Development processes like eXtreme Programming [13] treat refactoring as a key practice. Refactoring has revolutionized how programmers design software: it has enabled programmers to continuously explore the design space of large codebases, while preserving the existing behavior. Modern IDEs such as Eclipse, NetBeans, IntelliJ IDEA, or Visual Studio incorporate refactoring in their top menu and often compete on the basis of refactoring support.

Understanding the practice of refactoring helps researchers by validating or refuting assumptions that were previously based on folklore. It can also focus research attention on the refactorings that are popular in practice. Moreover, it can open new directions of research. For example, we discovered [101] that more than one third of the refactorings performed in practice are applied in a close time proximity to each other, thus forming a *cluster*. This result motivates new research [87, 131] into refactoring composition.

Since refactoring became an integral part of the daily activity of developers, they would benefit from better support for refactoring processes in IDEs. Let's consider a developer performs a refactoring manually. If a programming environment infers this refactoring *on-the-fly*, it will be able to better assist the developer. For example, the environment could check the correctness of the refactoring and advise the developer of any inconsistencies. Even if there are no problems, the environment could inform the developer about availability

of an automated refactoring, whose invocation would spare the developer from manual burden the next time he needs to perform this refactoring.

We infer refactorings from *continuous* code changes. Such inference is more accurate than the one performed on the VCS snapshots, since it does not suffer from the incompleteness and imprecision of the data stored in VCS (see Section 1.2). Moreover, continuous analysis allows us to correlate the inferred refactorings with other development activities, for example, invocations of automated refactorings. Additionally, inferring refactorings from continuous code changes can be performed on-the-fly and thus, enables an IDE to assist a developer with refactorings while they are in progress [44, 49].

In Section 1.5.1, we introduced our novel algorithm that continuously infers AST node operations — *adds*, *deletes*, and *updates* — from fine-grained code edits (e.g., typing characters). To infer refactorings, we designed and implemented an algorithm that takes as input these AST node operations. First, our refactoring inference algorithm infers high-level properties, e.g., replacing a variable reference with an expression. Then, from their combination it infers refactorings. For example, it infers that a local variable was inlined when it notices that a variable declaration is deleted and all its references are replaced with the initialization expression.

Our refactoring inference algorithm enabled us to perform a comparative study of manual and automated refactorings. We applied our algorithm on the real code evolution data from 23 developers, working in their natural environment for 1,520 hours. In our study, we answered seven research questions. In particular, we found that more than half of the refactorings were performed manually, and thus, the existing studies that focus on automated refactorings only might not be generalizable since they consider less than half of the total picture. We also found that the popularity of automated and manual refactorings differs. Our results present a fuller picture about the popularity of refactorings in general, which should help both researchers and tool builders to prioritize their work.

1.5.4 Identifying Previously Unknown Program Transformations

Although inferring refactorings benefits a developer in many ways, it is still limited, because refactorings represent only a single kind of program transformation that developers apply to their codebases. Besides, refactoring inference tools [31, 32, 44, 49, 53, 135, 137] support only a fixed number of predefined refactoring kinds, which narrows their usefulness even further.

We apply data mining techniques to detect previously *unknown* frequent code change patterns from a *continuous* sequence of code changes. Since our approach works on a continuous sequence of code changes, it can be applied either on-the-fly, or on a previously recorded sequence. An IDE that learns code change

patterns on-the-fly (i.e., as soon as the developer types them) can (i) perform the corresponding change automatically the next time a developer needs it or (ii) detect inconsistencies in the code changes if the developer continues to perform them manually.

There are unique challenges posed by our problem domain of program transformations, which render previous off-the-shelf data mining techniques [59] inapplicable. First, for program transformations, we need to mine a *contiguous sequence* of code changes that are ordered by their timestamps, without any previous knowledge of where the boundaries between patterns of transformations are. In contrast, standard data mining techniques operate on a database of transactions with well known boundaries (e.g., the user is checking out all items in the shopping cart). Thus, we need to divide the contiguous sequence of program changes into individual transactions. If we make the transactions disjoint (as in the existing data mining techniques), we will miss patterns that cross the boundaries of two transactions. To avoid missing such patterns, we employ *overlapping* transactions.

Second, unlike the standard frequent itemset mining, when mining frequent code change patterns, a high-level program transformation corresponding to a given pattern may contain several instances of the same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references of the renamed variable. Consequently, in our mining problem, a transaction may contain multiple instances of the same item kind, thus forming itembags rather than itemsets.

We designed and implemented a novel frequent code change pattern mining algorithm. Our algorithm employs the vertical data format [144] to directly access the transaction identifiers while computing new itemsets, which is crucial for effective handling of overlapping transactions and itembags. Our approach is inspired by several ideas from CHARM [145], the state-of-the-art algorithm for frequent closed itemsets mining that uses the vertical data format. In particular, our algorithm extends the notion of itemset-tidset tree (IT-tree) and adapts several optimization insights of CHARM.

We applied our novel algorithm on a large corpus of real world data that we employed in our previous study of refactoring practice [101], in which we accumulated 1,520 hours of code development from 23 developers working in their natural settings. Overall, our algorithm mined more than half a million item instances in less than six hours. We analyzed some of the frequent code change patterns detected by our algorithm and identified ten kinds of popular high-level program transformations. On average, 32% of the pattern occurrences reported by the algorithm led to high-level program transformation discoveries. Thus, our algorithm is scalable, effective, and useful.

1.6 Contributions

This dissertation makes the following contributions:

1. **Problem description.** This dissertation identifies important limitations of the current code development toolkits and traditional data sources for code evolution research.

2. **Notion of a change-oriented environment.** We present the notion of a development environment that focuses on code changes rather than code itself. Our approach fundamentally differs from the current state-of-the-art and overcomes its limitations.

3. **Change tracking infrastructure.** We developed an infrastructure that captures and replays all changes to the evolving code as well as many other development actions. Our infrastructure represents code changes uniformly and consistently as AST node operations. Our infrastructure serves as the foundation of a change-oriented environment.

4. **Inference of high-level program transformations.** We employed our change tracking infrastructure to design and implement two algorithms that infer high-level program transformations from low-level code edits. One algorithm detects known transformations like refactorings with a high accuracy, while another helps identify *previously unknown* program transformations. Our algorithms employ *continuous* change analysis, and thus, can be applied both *online* and *offline*.

5. **Empirical studies.** We performed empirical studies on code evolution and practice of refactoring and answered 12 research questions that could not be answered without an infrastructure like ours.

1.7 Organization

The rest of this dissertation is organized as follows. Chapter 2 presents the infrastructure of our change-oriented development environment. The first part introduces our tool, CodingTracker, that captures and replays a variety of development actions. The second part presents our algorithm for inferring AST node operations from low-level text edits. The chapter concludes with the results of our empirical study on source code evolution.

Chapter 3 presents our algorithm that infers known high-level program transformations, e.g., refactorings, from low-level program transformations (AST node operations) using continuous analysis. The second part of the chapter presents the results on our comparative study of manual vs. automated refactoring practice.

Chapter 4 presents our data mining algorithm for detecting *previously unknown* high-level program transformations. The chapter concludes with an empirical evaluation of our algorithm on real-world code evolution data that shows that our algorithm is scalable, effective, and useful.

Chapter 5 overviews related work and Chapter 6 presents future work. Chapter 7 concludes this dissertation.

Parts of this dissertation have been published in conference papers and technical reports: Chapter 2 is presented in an ECOOP-2012 paper [104], Chapter 3 is described in an ECOOP-2013 paper [101], and Chapter 4 is currently under submission [102] to a top-tier software engineering conference. All these chapters have been reorganized and augmented for inclusion in this dissertation.

Chapter 2

Tracking Fine-Grained and Precise Changes to the Code

2.1 Capturing and Replaying Code Changes

Our goal is to capture and replay code changes in a way that reproduces the actual development sessions as closely as possible. Note that capturing only code changing events is not sufficient for our goal. In particular, there are a number of code development events that define a context for code changing events. Without capturing these related events, it is impossible to correctly replay code changes. Therefore, besides code changing events, we track such related events as editing a file, saving a file, closing a file (with or without saving its content), moving a resource, etc. Some code development events are not directly related to code changes, but they still are part of code evolution, and thus, we track them to get a better insight into development practices. Examples of such events are test runs and interactions with a Version Control System (VCS). Table 2.1 presents the complete list of the registered events, which we split into 38 different kinds and grouped in 10 categories.

To keep track of code development events, we developed an Eclipse plug-in, CodingTracker. CodingTracker records the detailed information about each registered event, including the timestamp at which this event is triggered. For example, for a performed/undone/redone text edit, CodingTracker records the offset of the edit in the edited document, the removed text (if any), and the added text (if any). In fact, the recorded information is so detailed and precise that CodingTracker's replayer uses it to reconstruct the state of the evolving code at any point in time.

CodingTracker's replayer is an Eclipse View that is displayed alongside other Views of an Eclipse workbench, thus enabling a developer to see the results of the replayed events in the same Eclipse instance. Figure 2.1 illustrates CodingTracker's replayer. The replayer allows a developer to load a recorded sequence of events, browse it, hide events of the kinds that the developer is not interested in, and replay the sequence at any desired pace. Additionally, the replayer supports breakpoints (the row labeled with number 3 in Figure 2.1), which cause the replaying to stop after reaching the corresponding event. The row of an event that is about to be replayed is labeled with number 2. A developer can select any event in the list (the row

Category	Event	Description
Text editing	Perform/Undo/Redo text edit	Perform/Undo/Redo a text edit in a Java editor.
	Perform/Undo/Redo compare editor text edit	Perform/Undo/Redo a text edit in a compare editor.
File editing	Edit file	Start editing a file in a Java editor.
	Edit unsynchronized file	Start editing a file in a Java editor that is not synchronized with the underlying resource.
	New file	A file is about to be edited for the first time.
	Refresh file	Refresh a file in a Java editor to synchronize it with the underlying resource.
	Save file	Save file in a Java editor.
	Close file	Close file in a Java editor.
Compare editors	Open compare editor	Open a new compare editor.
	Save compare editor	Save a compare editor.
	Close compare editor	Close a compare editor.
Refactorings	Start refactoring	Perform/Undo/Redo a refactoring.
	Finish refactoring	A refactoring is completed.
Resource manipulation	Create resource	Create a new resource (e.g. file).
	Copy resource	Copy a resource to a different location.
	Move resource	Move a resource to a different location.
	Delete resource	Delete a resource.
	Externally modify resource	Modify a resource from outside of Eclipse (e.g. using a different text editor).
Interactions with Version Control System (VCS)	CVS/SVN update file	Update a file from VCS.
	CVS/SVN commit file	Commit a file to VCS.
	CVS/SVN initial commit file	Commit a file to VCS for the first time.
JUnit test runs	Launch test session	A test session is about to be started.
	Start test session	Start a test session.
	Finish test session	A test session is completed.
	Start test case	Start a test case.
	Finish test case	A test case is completed.
Start up events	Launch application	Run/Debug the developed application.
	Start Eclipse	Start an instance of Eclipse.
Workspace and Project Options	Change workspace options	Change global workspace options.
	Change project options	Change options of a refactored project.
Project References	Change referencing projects	Change the list of projects that reference a refactored project.

Table 2.1: The complete list of events recorded by CodingTracker.

labeled with number 1) and see its detailed information in the bottom pane. CodingTracker's replayer uses red, yellow, and blue colors to mark breakpoints, the event that is about to be replayed, and the selected event correspondingly.

To ensure that the recorded data is correct, CodingTracker records redundant information for some events. This additional data is used to check that the reconstructed state of the code matches the original one. For example, for every text edit, CodingTracker records the removed text (if any) rather than just the length of the removed text, which would be sufficient to replay the event. For CVS/SVN commits, CodingTracker records the whole snapshot of the committed file. While replaying text edits and CVS/SVN commits, CodingTracker checks that the edited document indeed contains the removed text and the committed file matches its captured snapshot[1].

Eclipse creates a refactoring descriptor for every invocation of automated refactorings. Refactoring descriptors are designed to capture sufficient information to enable replaying of the corresponding refactorings. Nevertheless, we found that some descriptors do not store important refactoring configuration options and thus, cannot be used to reliably replay the corresponding refactorings. For example, the descriptor of Extract Method refactoring does not capture information about the extracted method's parameters. Consequently, an Extract Method refactoring cannot be correctly reproduced from a descriptor recorded by Eclipse. To overcome this issue, besides recording refactoring descriptors of the performed/undone/redone automated Eclipse refactorings, CodingTracker records the refactorings' effects on the underlying code. A refactoring's effects are events triggered by the execution of this refactoring — usually, one or more events from the *Text editing*, *File editing*, and *Resource manipulation* categories presented in Table 2.1. In a sequence of recorded events, effects of an automated refactoring are located between its *Start refactoring* and *Finish refactoring* events as shown in Figure 2.1. To ensure robust replaying, CodingTracker replays the recorded refactorings' effects rather than their descriptors.

2.2 Inferring AST Node Operations

2.2.1 Unit of Code Change

Individual code edits collected by CodingTracker are too low-level and irregular to express well the intents of a developer, who reasons in terms of program entities rather than text characters or lines. Therefore, we define a unit of code change as a higher level of abstraction — an atomic operation on an Abstract Syntax Tree (AST) node: *add*, *delete*, or *update*, where *add* adds a node to AST, *delete* removes a node from AST,

[1]Note that replaying CVS/SVN commits does not involve any interactions with VCS, but rather checks the correctness of the replaying process.

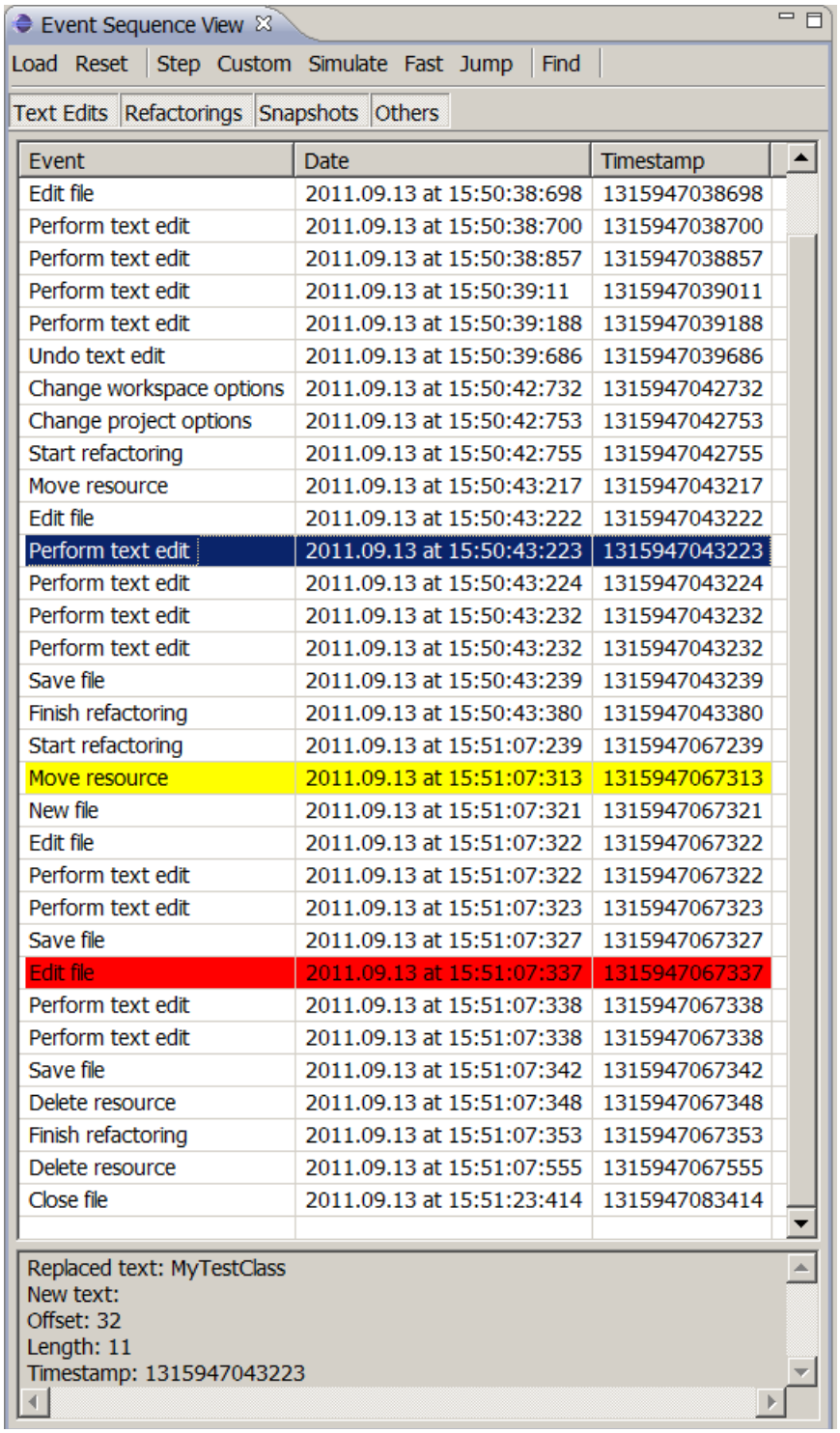


Figure 2.1: CodingTracker's replayer with a preloaded sequence of events.

and *update* changes a property of an existing AST node (e.g. name of a variable). Note that although AST node operations represent changes more uniformly than individual code edits, they are still too low-level to be directly useful for a developer. In our program transformations inference infrastructure, these AST node operations form the lowest level, from which we infer high-level program transformations (e.g., refactorings — see Section 3.2).

We infer AST node operations from code edits collected by CodingTracker. We apply our inferencing algorithm while replaying the collected data. CodingTracker's replayer makes it possible to reproduce the recorded development sessions, which helped us to improve our AST node operations inferencing algorithm to effectively handle the variety of code development scenarios recorded by CodingTracker.

AST node operations are the lowest level of program transformations. A high-level transformation might involve several AST node operations. For example, a Rename Local Variable refactoring involves as many *update* AST node operations as the number of bindings of the renamed local variable. To reflect this structural hierarchy, we need to establish how AST node operations correlate with other actions of a developer, e.g. whether an AST node operation is a result of an automated refactoring. Therefore, CodingTracker inserts the inferred AST node operations in the original event sequence right after the subsequence of code edits that produce them.

Figure 2.2 illustrates the same event sequence as Figure 2.1, but with the inferred AST node operations. We can see the details about the selected AST operation in the bottom pane. For example, this operation changes the node's text from `MyTestClass` to `MyClass`. Another observation is that the displayed sequence contains several instances of *AST file operation*, which is an equivalent of *Edit file* event for AST node operations. That is, instead of recording the source code file that an AST node operation belongs to in every such operation, we factor this information out into a separate *AST file operation*, which tracks the source code file of all subsequent AST node operations.

2.2.2 Inferencing Algorithm

Given an edited document, a single text edit is fully described by a 3-tuple `(<offset>, <removed text length>, <added text>)`, where `<offset>` is the offset of the edit in the edited document, `<removed text length>` is the length of the text that is removed at the specified offset, and `<added text>` is the text that is added at the specified offset. If `<removed text length>` is 0, the edit does not remove any text. If `<added text>` is empty, the edit does not add any text. If `<removed text length>` is not 0 and `<added text>` is not empty, the edit replaces the removed text with the `<added text>` in the edited document.

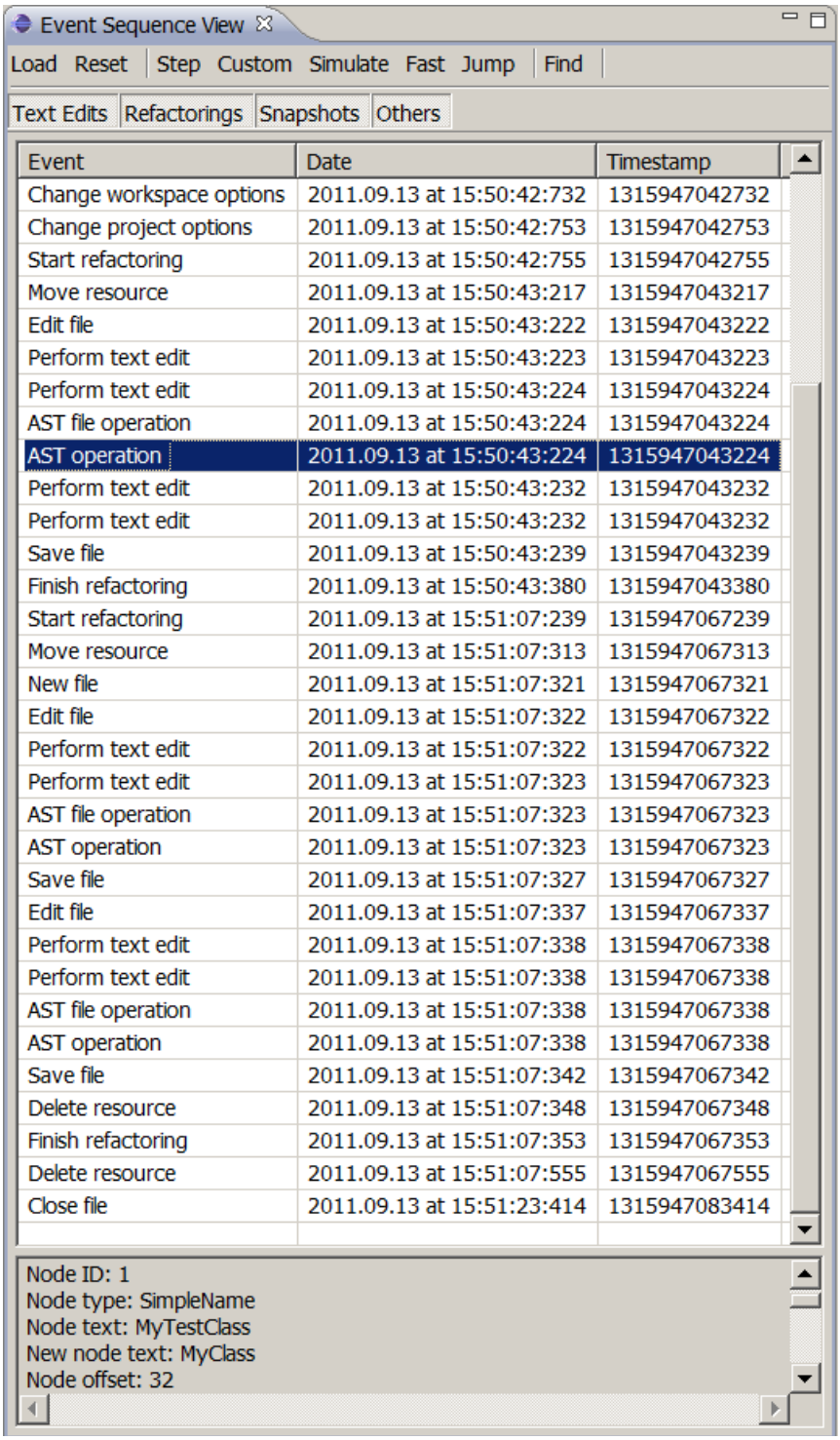
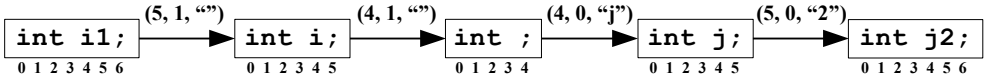
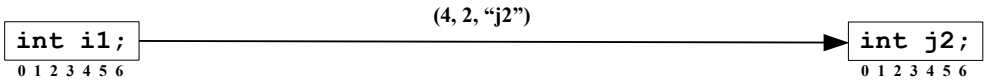


Figure 2.2: CodingTracker's replayer with a preloaded sequence of events, which is the same as in Figure 2.1, except that it contains the inferred AST node operations. Note how the inferred operations appear right after the text edits that produce them.



(a) A sequence of text edits recorded by CodingTracker for renaming variable `i1` to `j2`.



(b) A *coherent text edit* that “glues” together all text edits shown in Figure 2.3(a).

Figure 2.3: An example of changing a variable's name represented both as individual text edits recorded by CodingTracker (Figure 2.3(a)) and as a single *coherent text edit* (Figure 2.3(b)). Each box shows the content of the edited document. The offset of every document's character is shown under each box. The 3-tuples describing text edits are shown above the arrows that connect boxes.

Our inferencing algorithm converts text edits into operations on the corresponding AST nodes. First, the algorithm assigns a unique ID to every AST node in the old AST. Next, the algorithm considers the effect of each text edit on the position of the node in the new AST in order to match the old and the new AST nodes. The matched nodes in the new AST get their IDs from their counterparts in the old AST. If the content of a matched AST node has changed, the algorithm generates the corresponding *update* operation. The algorithm generates a *delete* operation for every unmatched node in the old AST and an *add* operation for every unmatched node in the new AST, assigning it a unique ID.

In the following, we describe several heuristics that improve the precision of our inferencing algorithm. Then, we explain our algorithm in more detail and demonstrate it using an example.

Gluing

Figure 2.3(a) illustrates an example of text edits produced by a developer, who renamed variable `i1` to `j2` by first removing the old name using backspace and then typing in the new name. This single operation of changing a variable's name involves 4 distinct text edits that are recorded by CodingTracker. At the same time, all these text edits are so closely related to each other that they can be “glued” together into a single text edit with the same effect on the underlying text, which is shown in Figure 2.3(b). We call such “glued” text edits *coherent text edits* and use them instead of the original text edits recorded by CodingTracker in our AST node operations inferencing algorithm. This drastically reduces the number of inferred AST node operations and makes them better represent the intents of a developer.

Intuitively, a coherent text edit combines two or more individual text edits that are consecutive both in time and textual position. More precisely, a text edit `e2` is “glued” to a preceding text edit `e1`, when:

1. `e2` immediately follows `e1`, i.e., there is no other events between these two text edits.

2. `e2` continues the text change of `e1`, i.e., text edit `e2` starts at the offset, at which `e1` stopped.

Note that in the above heuristics `e1` can be either a text edit recorded by CodingTracker or a *coherent text edit*, produced from "gluing" several preceding text edits, while `e2` is a text edit recorded by CodingTracker.

Linked Edits

Eclipse offers a code editing feature that allows simultaneous editing of a program entity in all its bindings that are present in the opened document. Each binding of the edited entity becomes an edit box and every text edit in a single edit box is immediately reflected in all other boxes as well. This feature is often used to rename program entities, in particular to name extracted methods. Since text edits in a single edit box are intermixed with the corresponding edits in other edit boxes, to apply our "gluing" heuristics presented above, we treat edits in each box disjointly, constructing a separate *coherent text edit* for every edit box. When a boxed edit is over, the constructed coherent text edits are processed one by one to infer the corresponding AST node operations.

Jumping over Unparsable Code

The AST node operations inferencing algorithm processes coherent text edits as soon as they are constructed, except when text edits introduce parsing errors in the underlying code. Parsing errors might confuse the parser that creates ASTs for our algorithm, which could lead to imprecise inferencing results. Therefore, when a text edit breaks the AST, the algorithm postpones inferencing until the AST is well-formed again. Such postponement causes accumulation of several coherent text edits, which are processed by our inferencing algorithm together. Figure 2.4 shows an example of code editing scenario that requires inference postponing. A developer inserts brackets around the body of an `if` statement. The first coherent text edit adds an opening bracket, breaking the structure of the AST, while the second coherent text edit adds a closing bracket, bringing the AST back to a well-formed state. The inference is postponed until the second edit fixes the AST. Note that sometimes we have to apply the inferencing algorithm even when the underlying program's AST is still broken, for example, when a developer closes the editor before fixing the AST. This could lead to some imprecision in the inferred AST node operations, but our experience suggests that such scenarios are rare in practice. In particular, only 0.4% of the inferencing algorithm applications on the data that we collected during our empirical study on code evolution (see Section 2.3) were performed when the edited code contained parsing errors.

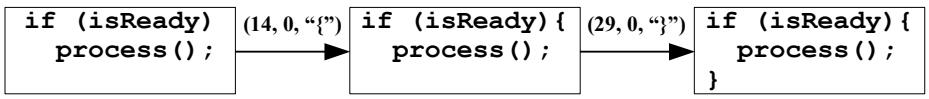


Figure 2.4: An example of two code edits: the first edit breaks the AST of the edited program, while the second edit brings the AST back to a well-formed state.

input: *oldAST*, *newAST*, *cteList* // the list of coherent text edits
output: *astNodeOperations*
1 *astNodeOperations* = ∅;
2 *matchedNodes* = ∅;
3 *coveringPosition* = getCommonCoveringNodePosition(*oldAST*, *newAST*, *cteList*);
4 *oldCoveringNode* = getNode(*oldAST*, *coveringPosition*);
5 *newCoveringNode* = getNode(*newAST*, *coveringPosition*);
6 matchOutliers(*cteList*, *oldCoveringNode*, *newCoveringNode*, *matchedNodes*);
7 matchSamePositionNodes(*oldAST*, *newAST*, *oldCoveringNode*, *newCoveringNode*, *matchedNodes*);
8 createASTNodeOperations(*oldCoveringNode*, *newCoveringNode*, *matchedNodes*, *astNodeOperations*);

Figure 2.5: Overview of our AST node operations inferencing algorithm.

Pseudocode

Figure 2.5 shows an overview of our AST node operations inferencing algorithm. The algorithm takes as input the list of *coherent text edits*, *cteList*, the AST of the edited code before the text edits, *oldAST*, and the AST after the edits, *newAST*. The output of the algorithm is an unordered collection of the inferred AST node operations.

The inferencing algorithm is applied as soon as a new coherent text edit is completed, unless the underlying code is unparsable, in which case the inferencing is postponed until the code becomes parsable again. As long as the code remains parsable, *cteList* contains a single coherent text edit. If the code becomes unparsable for some time, *cteList* will contain the accumulated coherent text edits that bring the code back into the parsable state. Note that *newAST* represents the code that is a result of the edits in *cteList* applied to the code represented by *oldAST*. Since replaying the edits in *cteList* is not a part of the inferencing algorithm, we supply both *oldAST* and *newAST* as the algorithm's inputs.

Each inferred operation captures the persistent ID of the affected AST node. Persistent IDs uniquely identify AST nodes in an application throughout its evolution. Note that given an AST, a node can be identified by its *position* in this AST. A node's position in an AST is the traversal path to this node from the root of the AST. Since the position of an AST node may change with the changes to its AST, we assign a unique persistent ID to every AST node and keep the mapping from positions to persistent IDs, updating it accordingly whenever a node's position is changed as a result of code changes.

Most of the time, edits in *cteList* affect only a small part of the code's AST. Therefore, the first step of

```
procedure: matchOutliers(cteList, oldCoveringNode, newCoveringNode, matchedNodes)
1  foreach (oldNode ∈ getDescendants(oldCoveringNode)) {
2    deltaOffset = 0;
3    foreach (textEdit ∈ cteList) {
4      if (affects(textEdit, oldNode, deltaOffset) {
5        continue foreach_line1;
6      } else {
7        deltaOffset += getEditOffset(textEdit, oldNode, deltaOffset);
8      }
9    }
10   if (∃ newNode ∈ getDescendants(newCoveringNode) :
         getOffset(oldNode) + deltaOffset == getOffset(newNode) &&
         haveSameASTNodeTypes(oldNode, newNode)) {
11     matchedNodes ∪= (oldNode, newNode);
12   }
13 }
```

Figure 2.6: This procedure matches AST nodes that are not affected by text edits (i.e., these nodes are outliers in respect to the edits).

the algorithm (lines 3 – 5) establishes the root of the changed subtree – a *common covering node* that is present in both the old and the new ASTs and completely encloses the edits in *cteList*. To find a common covering node, we first look for a local covering node in *oldAST* and a local covering node in *newAST*. These local covering nodes are the innermost nodes that fully encompass the edits in *cteList*. The common part of the traversal paths to the local covering nodes from the roots of their ASTs represents the position of the common covering node (assigned to *coveringPosition* in line 3).

Next, the algorithm matches AST nodes that are not affected by the edits in *cteList* (line 6). Then, the inferencing algorithm matches yet unmatched nodes by their AST node types and positions (line 7). Finally, the algorithm creates the corresponding AST node operations for the matched and unmatched AST nodes (line 8).

Figure 2.6 shows the details of matching AST nodes that are not affected by the edits in *cteList*. Every descendant node of the common covering node in the old AST is checked against the edits in *cteList* (lines 1 – 13). An edit does not affect a node if the code that this node represents is either completely before the edited code fragment or completely after it. If a node's code is completely before the edited code fragment, the edit does not impact the node's offset. Otherwise, the edit shifts the node's offset with `<added text length> - <removed text length>`. These shifts are calculated by *getEditOffset* and accumulated in *deltaOffset* (line 7). If no edits affect a node, the algorithm looks for its matching node in the new AST (line 10). Every matched pair of nodes is added to *matchedNodes* (line 11).

Figure 2.7 shows how our algorithm matches nodes that have the same AST node type and the same

```
procedure: matchSamePositionNodes(oldAST,newAST,oldCoveringNode,newCoveringNode,matchedNodes)
1  foreach (oldNode ∈ getDescendants(oldCoveringNode) : oldNode ∉ getOldNodes(matchedNodes)) {
2    oldPosition = getNodePositionInAST(oldNode, oldAST);
3    newNode = getNode(newAST, oldPosition);
4    if (∃ newNode ∈ getDescendants(newCoveringNode) : haveSameASTNodeTypes(oldNode, newNode) {
5      matchedNodes ∪= (oldNode, newNode);
6    }
7  }
```

Figure 2.7: This procedure matches AST nodes by their types and positions.

```
procedure: createASTNodeOperations(oldCoveringNode,newCoveringNode,matchedNodes,astNodeOperations)
1  foreach ((oldNode, newNode) ∈ matchedNodes) {
2    if (getText(oldNode) ≠ getText(newNode)) {
3      astNodeOperations ∪= getUpdateOperation(oldNode, newNode);
4    }
5  }
6  foreach (oldNode ∈ getDescendants(oldCoveringNode) : oldNode ∉ getOldNodes(matchedNodes)) {
7    astNodeOperations ∪= getDeleteOperation(oldNode);
8  }
9  foreach (newNode ∈ getDescendants(newCoveringNode) : newNode ∉ getNewNodes(matchedNodes)) {
10   astNodeOperations ∪= getAddOperation(newNode);
11 }
```

Figure 2.8: This procedure creates AST node operations for matched and unmatched nodes.

position in the old and the new ASTs. Note that at this step, the algorithm considers only yet unmatched nodes (line 1).

Figure 2.8 shows how the algorithm creates an *update* operation for every matched node whose content has changed (lines 1 – 5), a *delete* operation for every unmatched node in the old AST (lines 6 – 8), and an *add* operation for every unmatched node in the new AST (lines 9 – 11).

Example

Figure 2.9 illustrates a coherent text edit that changes a variable declaration. Figure 2.10 demonstrates the inferred AST node operations for this edit. Connected ovals represent the nodes of the old and the new ASTs. Dashed arrows represent the inferred operations. Labels above the arrows indicate the kind of the corresponding operations.

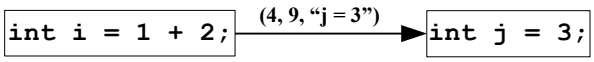


Figure 2.9: An example of a text edit that changes a variable declaration.

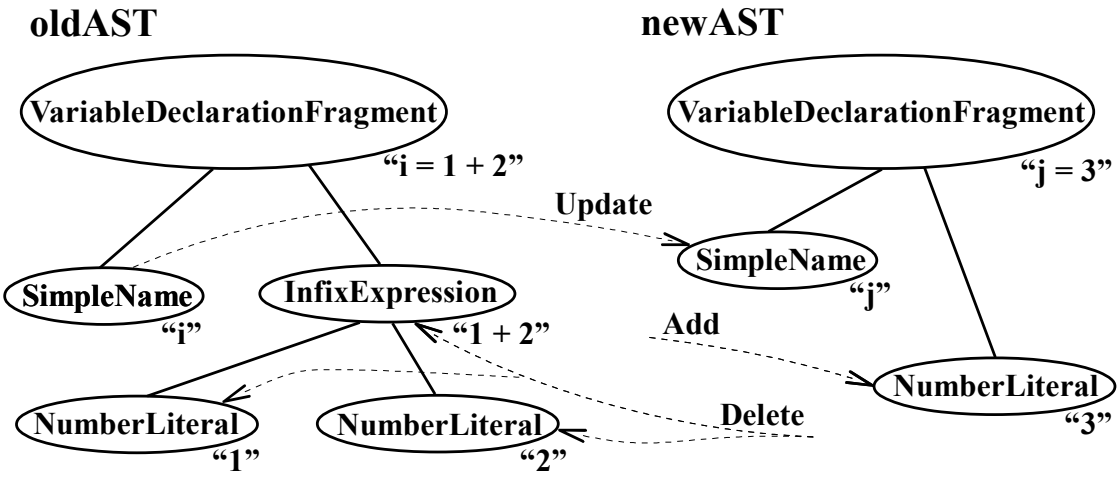


Figure 2.10: The inferred AST node operations for the text edit in Figure 2.9.

2.3 Empirical Study of Code Evolution

2.3.1 Introduction

To show the benefits of capturing more detailed data about code evolution as well as quantify the limitations of VCS snapshots, we performed a field study on 24 developers working in their natural settings. Every participant of our study installed CodingTracker in his/her Eclipse IDE. Overall, we collected data for 1,652 hours of development, which involve 2,000 commits comprising 23,002 committed files, and 9,639 test session runs involving 314,085 testcase runs.

The collected data enables us to answer five research questions:

RQ1: How much code evolution data is not stored in VCS?

RQ2: How much do developers intersperse refactorings and edits in the same commit?

RQ3: How frequently do developers fix failing tests by changing the test itself?

RQ4: How many changes are committed to VCS without being tested?

RQ5: What is the temporal and spacial locality of changes?

We found that 37% of code changes are *shadowed* by other changes, and are not stored in VCS. Thus, VCS-based code evolution research is incomplete. Second, programmers intersperse different kinds of changes

Number of participants	Programming Experience (years)
1	1 - 2
4	2 - 5
11	5 - 10
6	> 10

Table 2.2: Programming experience of the participants.

in the same commit. For example, 46% of refactored program entities are also edited in the same commit. This overlap makes the VCS-based research imprecise. The data collected by CodingTracker enabled us to answer research questions that could not be answered using VCS data alone. The data reveals that 40% of test fixes involve changes to the tests, which motivates the need for automated test fixing tools [28, 29, 97]. In addition, 24% of changes committed to VCS are untested. This shows the usefulness of continuous integration tools [6, 9, 68, 70]. Finally, we found that 85% of changes to a method during an hour interval are clustered within 15 minutes. This shows the importance of novel IDE user interfaces [15] that allow developers to focus on a particular part of the system.

2.3.2 Research Methodology

To answer our research questions, we conducted a user study on 24 participants. We recruited 11 professional developers who worked on projects in different domains, including marketing, banking, business process management, and database management. We also recruited 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at Urbana-Champaign. Table 2.2 shows the programming experience of our participants[2]. In the course of our study, we collected code evolution data for 1,652 hours of code development with a mean of 69 hours and a standard deviation of 62 hours.

To collect code evolution data, we asked each participant to install CodingTracker plug-in in his/her Eclipse IDE. During the study, CodingTracker recorded a variety of evolution data at several levels ranging from individual code edits up to high-level events like automated refactoring invocations, test runs, and interactions with Version Control System (VCS). CodingTracker employed an existing infrastructure [132] to regularly upload the collected data to our centralized repository.

To infer AST node operations from the collected raw edits, we apply our inferencing algorithm described in Section 2.2. Our research questions require establishing how AST node operations correlate with different developer's actions, e.g., whether an AST operation is a result of a refactoring, whether AST operations are followed by a commit or preceded by tests, etc. Therefore, CodingTracker inserts the inferred AST node

[2]Note that only 22 out of 24 participants filled the survey and specified their programming experience.

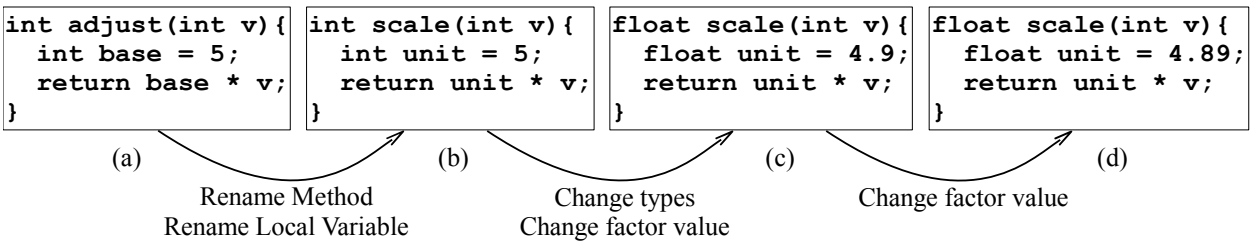


Figure 2.11: A code evolution scenario that illustrates a shadowed code change and an overlap of refactorings with other code changes.

operations in the original event sequence right after the subsequence of code edits that produce them. We answer every research question by processing the output of the inferencing algorithm with the question-specific analyzer.

2.3.3 Results

RQ1: How much code evolution data is not stored in VCS?

A single VCS commit may contain hours or days worth of development. During this period, a developer may change the same code fragment multiple times, with the latter changes *shadowing* the former changes.

Figure 2.11 presents a code evolution scenario. The developer checks out the latest revision of the code shown in Figure 2.11(a) and then applies two refactorings and performs several code changes to a single method, `adjust`. First, the developer renames the method and its local variable, `base`, giving them intention-revealing names. Next, to improve the precision of the calculation, the developer changes the type of the computed value and the production factor, `unit`, from `int` to `float` and assigns a more precise value to `unit`. Last, the developer decides that the value of `unit` should be even more precise, and changes it again. Finally, the developer checks the code shown in Figure 2.11(d) into the VCS.

Note that in the above scenario, the developer changes the value assigned to `unit` twice, and the second change *shadows* the first one. The committed snapshot shown in Figure 2.11(d) does not reflect the fact that the value assigned to `unit` was gradually refined in several steps, and thus, some code evolution information is lost.

To quantify the extent of code evolution data losses in VCS snapshots, we calculate how many code changes never make it to VCS. We compute the total number of changes that happen between each two commits of a source code file and the number of changes that are *shadowed*, and thus, do not reach VCS. We get the number of *reaching* changes by subtracting the number of shadowed changes from the total number of changes. To recall, a unit of code change is an *add*, *delete*, or *update* operation on an AST node. For any

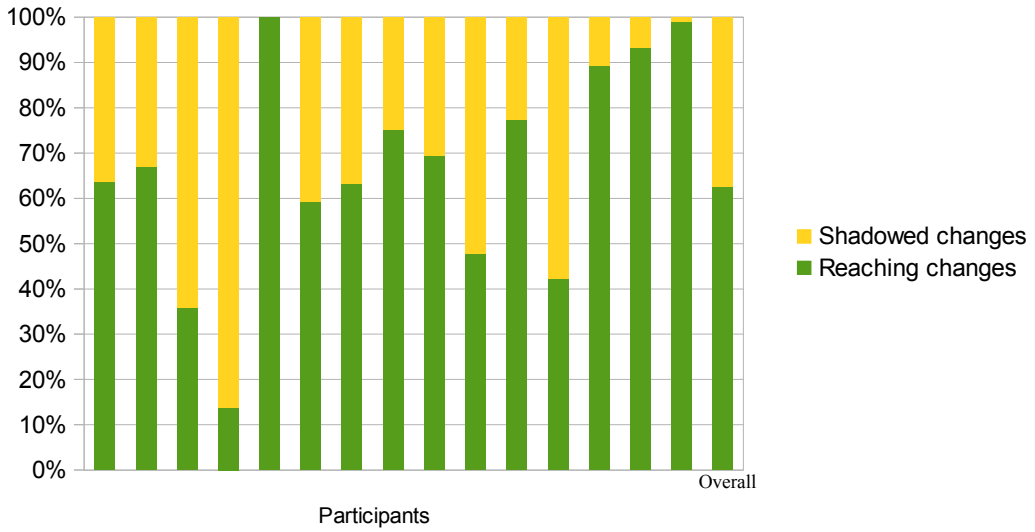


Figure 2.12: Ratio of reaching and shadowed changes.

two operations on the same AST node, the second operation always shadows the first one. Additionally, if an AST node is both added and eventually deleted before being committed, then all operations that affect this node are shadowed, since no data about this node reaches the commit.

Figure 2.12 shows the ratio of reaching and shadowed changes for our participants. Note that we recorded interactions with VCS only for 15 participants who used Eclipse-based VCS clients. A separate bar presents the data for each such participant. The last bar presents the aggregated result. Overall, we recorded 2,000 commits comprising 23,002 committed files.

The results in Figure 2.12 demonstrate that on average, 37% of changes are shadowed and do not reach VCS. To further understand the nature of shadowed code changes, we counted separately those shadowed changes that are commenting/uncommenting parts of the code or undoing some previous changes. If a change is both commenting/uncommenting and undoing, then it is counted as commenting/uncommenting. Figure 2.13 presents the results. Overall, 78% of shadowed code changes are authentic changes, i.e., they represent actual changes rather than playing with the existing code by commenting/uncommenting it or undoing some previous changes.

Our results reveal that more than a third of all changes do not reach VCS and the vast majority of these lost changes are authentic. Thus, a code evolution analysis based on snapshots from VCS misses a significant fraction of important code changes, which could lead to imprecise results.

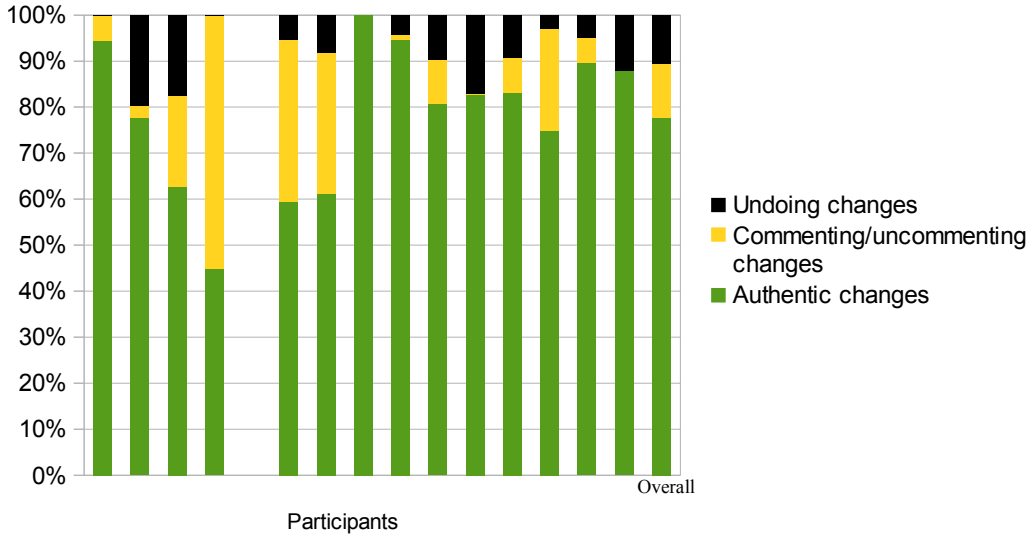


Figure 2.13: Composition of shadowed changes. The fifth bar is missing, since there are no shadowed changes for this participant.

RQ2: How much do developers intersperse refactorings and edits in the same commit?

Many tools [31, 32, 53, 79, 135, 137] compare a program's snapshots stored in VCS to infer the refactorings applied to it. As the first step, such a tool employs different similarity measures to match the refactored program entities in the two compared snapshots. Next, the tool uses the difference between the two matched program entities as an indicator of the kind of the applied refactoring. For example, two methods with different names but with similar code statements could serve as an evidence of a Rename Method refactoring [32]. If a refactored program entity is also changed in the same commit, both matching it across commits and deciding on the kind of refactoring applied to it become harder. Such code evolution scenarios undermine the accuracy of the snapshot-based refactoring inference tools.

Figure 2.11 shows an example of such a scenario. It starts with two refactorings, Rename Method and Rename Local Variable. After applying these refactorings, the developer continues to change the refactored entities – the body and the return type of the renamed method; the type and the initializer of the renamed local variable. Consequently, versions (a) and (d) in Figure 2.11 have so little in common that even a human being would have a hard time identifying the refactored program entities across commits.

To quantify how frequently refactorings and edits overlap, we calculate the number of refactored program entities that are also edited in the same commit. Our calculations employ the data collected by CodingTracker

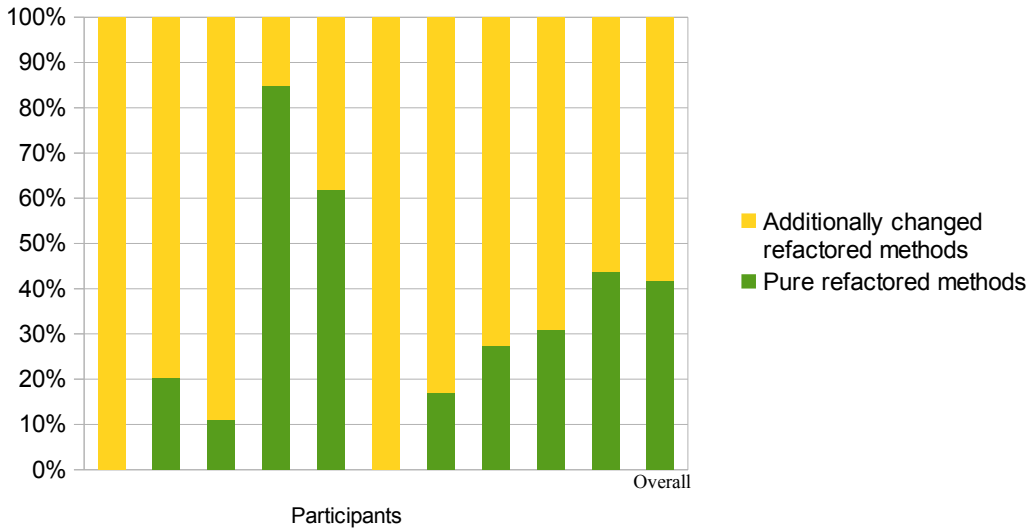


Figure 2.14: Ratio of purely refactored methods and those that are both refactored and additionally changed before being committed to VCS.

for ten participants who both used Eclipse-based VCS clients and performed automated refactorings. Note that we do not consider manual refactorings since they cannot be directly captured by our data collector, but rather need to be inferred from the collected data as an additional, non-trivial step.

First, we look at a single kind of program entities – methods. Figure 2.14 shows the ratio of those methods that are refactored only once before being committed (pure refactored methods) and those methods that are both refactored and edited (e.g., refactored more than once or refactored and edited manually) before being committed to VCS. We consider a method refactored/edited if either its declaration or any program entity in its body are affected by an automated refactoring/manual edit. Figure 2.14 shows that on average, 58% of methods are both refactored and additionally changed before reaching VCS.

Next, we refine our analysis to handle individual program entities. To detect whether two refactorings or a refactoring and a manual edit overlap, we introduce the notion of a *cluster* of program entities. For each program entity, we compute its cluster as a collection of closely related program entities. A cluster of a program entity includes this entity, all its descendants, its enclosing statement, and all descendants of its enclosing statement, except the enclosing statement's body and the body's descendants. For example, a cluster of a `StringLiteral` node inside a conditional expression of an `if` statement includes all nodes that are part of the conditional expression and the `if` statement itself, but does not include any nodes that are part of the `if` statement's body. We consider a program entity refactored/edited if any of the entities of its cluster is affected by an automated refactoring/manual edit. Figure 2.15 demonstrates that on average,

46% of program entities are both refactored and additionally changed in the same commit.

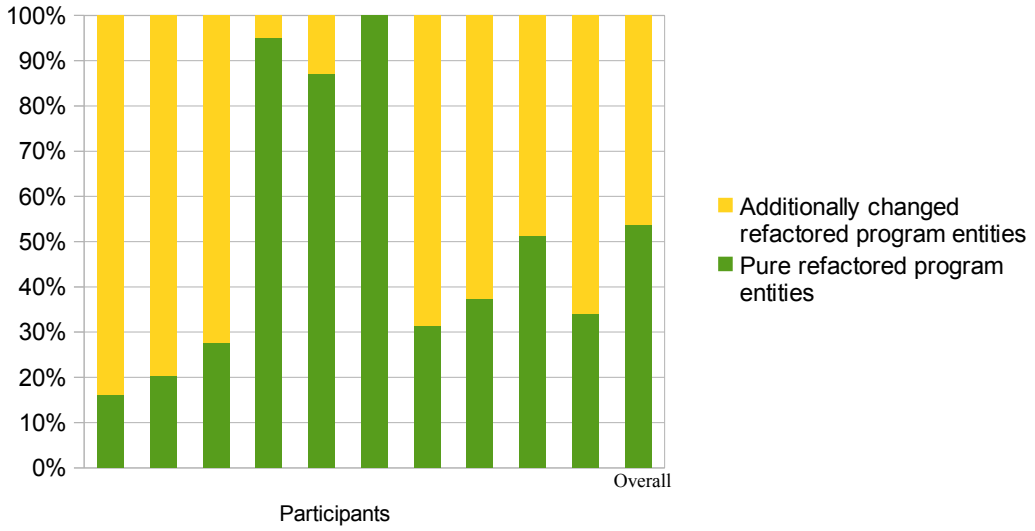


Figure 2.15: Ratio of purely refactored program entities and those that are both refactored and additionally changed before reaching a commit.

Our results indicate that most of the time, refactorings are tightly intermixed with other refactorings or manual edits before reaching VCS. This could severely undermine the effectiveness of refactoring inference tools that are based on VCS snapshots [31, 32, 53, 79, 135, 137].

RQ3: How frequently do developers fix failing tests by changing the test itself?

In response to ever-changing requirements, developers continuously add new features or adjust existing features of an application, which could cause some unit tests to fail. A test that fails due to the new functionality is considered *broken* since making it a passing test requires fixing the test itself rather than the application under test. Developers either have to fix the broken tests manually or use recent tools that can fix them automatically [28, 29, 97].

Figure 2.16 presents a unit test of a parser. This test checks that the parser produces a specific number of AST nodes for a given input. A new requirement to the system introduces an additional parsing rule. Implementing this new feature, a developer breaks this test, because the number of elements in the same input has changed. Thus, the developer needs to update the broken test accordingly.

We justify the need for automated test fixing tools by showing how often such scenarios happen in practice, i.e., how many failing tests are fixed by changing the test itself. We look for these scenarios in the data collected for 15 participants who ran JUnit tests as part of their code development process. Overall,

```
public void testElementsCount(){
  Parser p = new Parser();
  p.parse(getSource());
  assertEquals(p.getCount(), 5);
}
```

Figure 2.16: A unit test of a parser that checks the total number of elements in the parser's result.

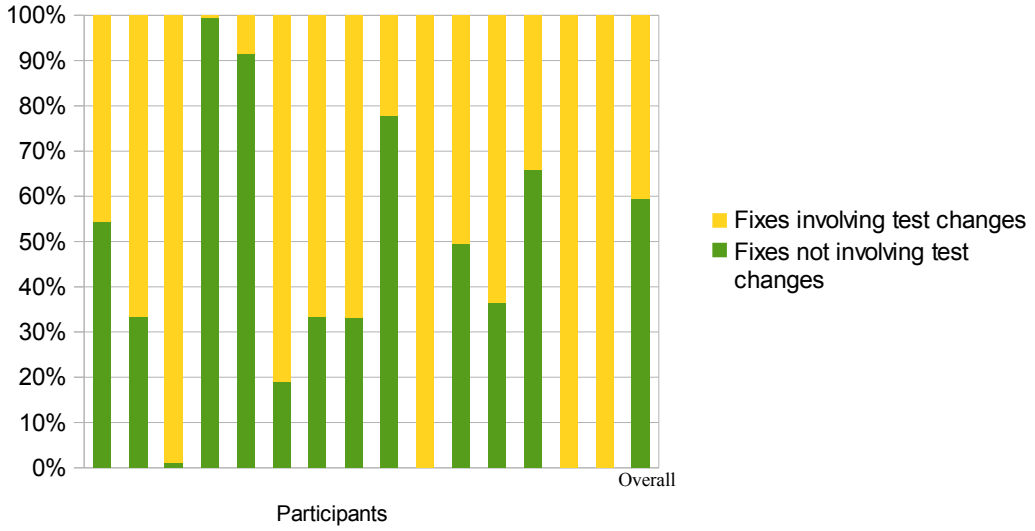


Figure 2.17: Ratio of test fixes involving and not involving changes to the tests.

we recorded 9,639 test session runs, involving 314,085 testcase runs. We track a failing test from the first run it fails until the run it passes successfully. Each such scenario is counted as a test fix. If a developer changes any classes in the test's package during this time span, we consider that fixing this failing test involves changing the test's code.

Figure 2.17 shows the ratio of test fixes involving and not involving changes to the tests. Our results show that on average, 40% of test fixes involve changes to the tests. Another observation is that every participant has some failing tests, whose fix requires changing them. Hence, a tool like ReAssert [29] could have benefited all of the participants, potentially helping to fix more than one third of all failing tests. Nevertheless, only a dedicated study would show how many of the required changes to the failing tests could be automated by ReAssert.

RQ4: How many changes are committed to VCS without being tested?

Committing untested code is considered a bad practice. A developer who commits untested code risks breaking the build and consequently, disrupting the development process. To prevent such scenarios and catch

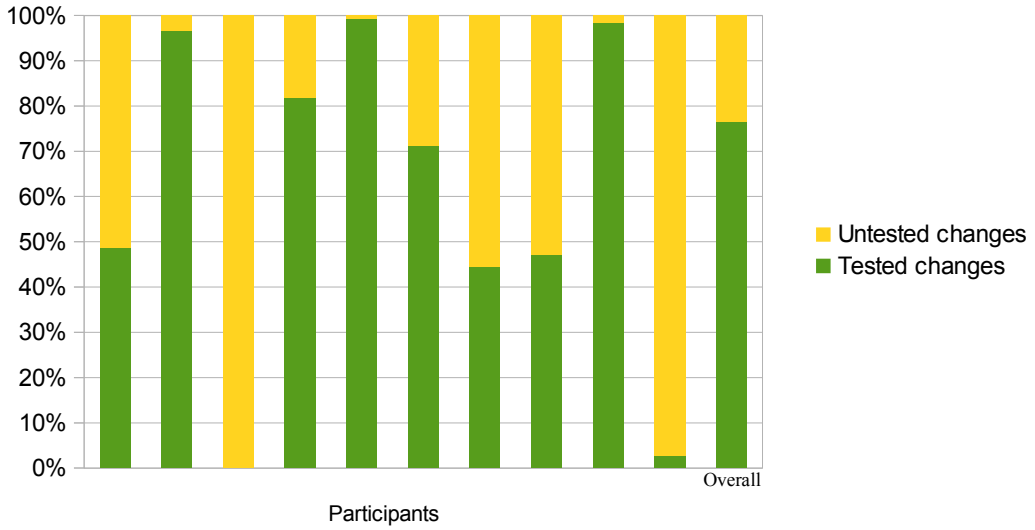


Figure 2.18: Ratio of tested and untested code changes that reach VCS.

broken builds early, the industry adopted continuous integration tools (e.g., Apache Gump [6], Bamboo [9], Hudson [68], and Jenkins [70]), which build and test every commit before integrating it into the trunk. Only those commits that successfully pass all the tests are merged into the trunk. Nevertheless, these tools are not yet pervasive. In particular, most projects that we studied did not employ any continuous integration tools. Therefore, we would like to quantitatively assess the necessity of such tools.

To assess the number of untested, potentially build-breaking changes that are committed to VCS, we measure how much developers change their code between tests and commits. Our measurements employ the data collected for ten participants who both used Eclipse-based VCS clients and ran JUnit tests. We consider each pair of consecutive commits of a source code file. If there are no test runs between these two commits, we disregard this pair of commits[3]. Otherwise, we count the total number of code changes that happen between these two commits. Also, we count all code changes since the last test run until the subsequent commit as *untested* changes. Subtracting the untested changes from the total number of changes between the two commits, we get the *tested* changes.

Figure 2.18 shows the ratio of tested and untested changes that reach VCS. Although the number of untested changes that reach a commit varies widely across the participants, every participant committed at least some untested changes. Overall, 24% of changes committed to VCS are untested. Figure 2.19 shows that 97% of the untested changes are authentic, i.e., we discard undos and comments.

[3]We are conservative in calculating the number of untested changes in order to avoid skewing our results with some corner case scenarios, e.g., when a project does not have automated unit tests at all (although this is problematic as well).

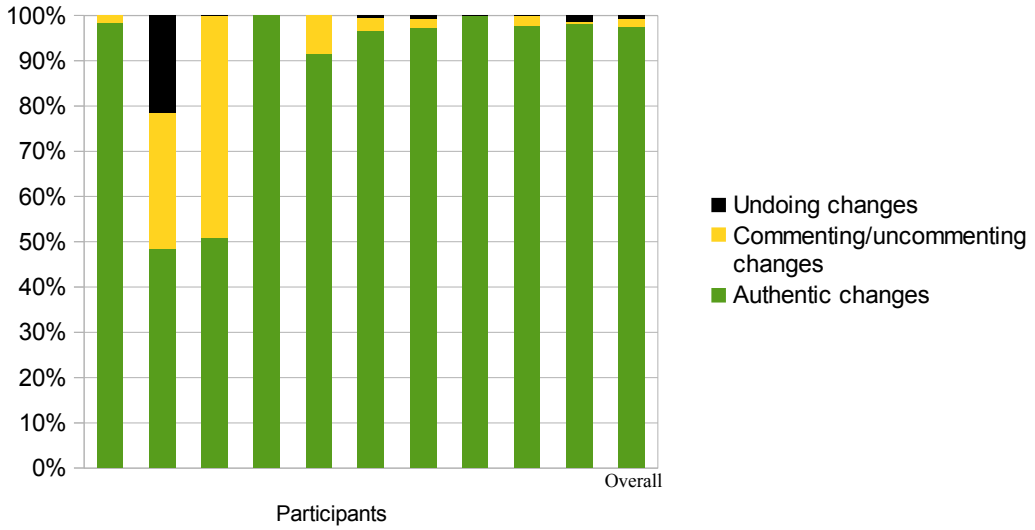


Figure 2.19: Composition of untested changes that reach VCS.

Note that even a small number of code changes may introduce a bug, and thus, break a build (unless the code is committed to a temporary branch). Besides, even a single developer with a habit to commit untested changes into the trunk may disrupt the development process of the entire team. Thus, our results confirm the usefulness of continuous integration tools, which ensure that all commits merged into the trunk are fully tested.

RQ5: What is the temporal and spacial locality of changes?

Simplifying the development process and increasing the productivity of a developer are among the major goals of an Integrated Development Environment (IDE). The better an IDE supports code changing behavior of a developer, the easier it is for him/her to develop the code. Code Bubbles [15] is an example of a state-of-the-art IDE with a completely reworked User Interface (UI). The novel UI enables a developer to concentrate on individual parts of an application. For example, a developer could pick one or more related methods that he/she is currently reviewing or editing and focus on them only.

To detect whether developers indeed focus their editing efforts on a particular method at any given point in time, we calculate the distribution of method-level code changes over time. We perform this calculation for all 24 participants who took part in our study, since it does not depend on any particular activity of the participant (e.g., interactions with VCS or test runs). We employ three sliding time windows spanning 15, 30, and 60 minutes. For every code change that happens in a particular method, we count the number

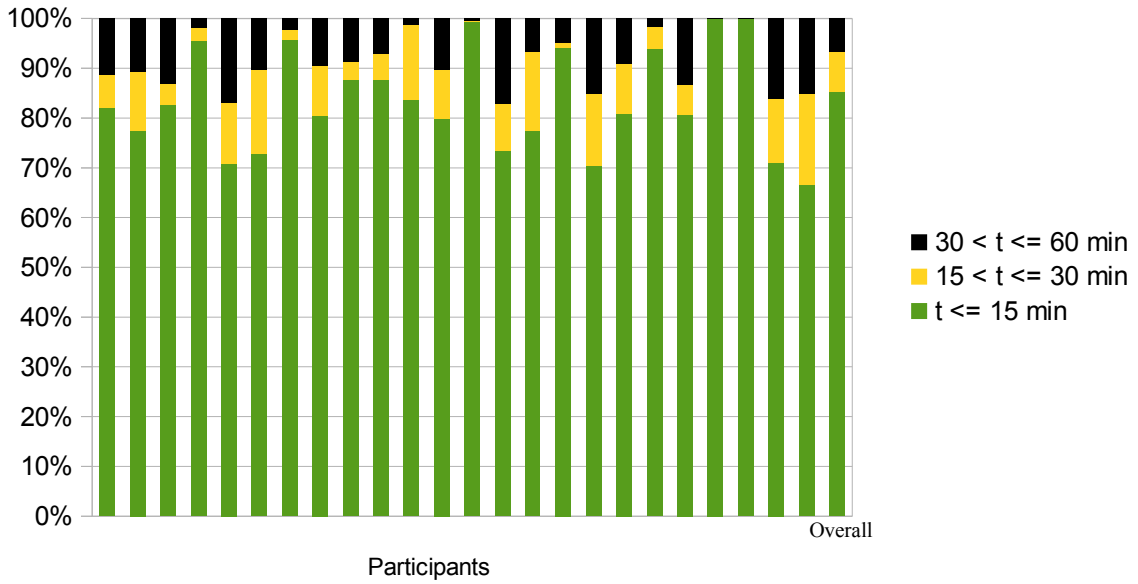


Figure 2.20: Ratio of method-level code changes for three sliding time windows: 15, 30, and 60 minutes.

of changes to this method within each sliding window, i.e., the number of changes 7.5 minutes, 15 minutes, and 30 minutes before and after the given change. Then, we sum the results for all code changes of each method. Finally, we add up all these sums for all the methods.

Figure 2.20 shows the ratio of method-level code changes for each of our three sliding time windows. On average, 85% of changes to a method during an hour interval are clustered within 15 minutes. Our results demonstrate that developers tend to concentrate edits to a particular method in a relatively small interval of time. The implication of this finding is that IDEs should provide visualizations of the code such that a programmer can focus on one method at a time.

2.3.4 Threats to Validity

There are several factors that might negatively impact the accuracy of our results. This section discusses the potential influence and possible mitigation for each of these factors.

Experimental Setup

Issues like privacy, confidentiality, and lack of trust in the reliability of research tools made it difficult to recruit developers to participate in our study. Therefore, we were unable to study a larger sample of experienced developers.

Many factors affect developers' practices. For example, developers may write code, refactor, test, and commit differently in different phases of software development, e.g., before and after a release. As another example, practices of developers who work in teams might be different than those who are the sole authors of their programs. Due to the uncontrolled nature of our study, it is not clear how such factors affect our results.

Our participants have used CodingTracker for different periods of time (See Section 2.3.2). Those participants who used CodingTracker more influenced our results more.

Our results are limited to developers who use Eclipse for Java programming because CodingTracker is an Eclipse plug-in that captures data about the evolution of Java code. However, we expect our results to generalize to similar programming environments.

AST Node Operations Inferencing Algorithm

To decide whether two individual text edits should be "glued" together, we apply certain heuristics, which are sufficient in most cases. Nevertheless, like any heuristics, they cannot cover all possible scenarios. As a result, our algorithm might infer multiple operations for a single change intended by a developer (e.g., a single rename of a variable). This artificial increase in the number of AST node operations can potentially skew the results for each question. However, our personal experience shows that such corner case scenarios are infrequent in practice and thus, their influence on our results is negligible.

In this study, we did not infer the *move* operation, but rather represented the corresponding action as *delete* followed by *add*. Consequently, the number of AST node operations that our data analyzers operate on might be inflated. At the same time, all our results are computed as ratios of the number of operations, which substantially diminishes the effect of this inflation.

Although our AST node operations inferencing algorithm does not expect that the underlying code is always parsable, it produces the most precise results for a particular subsequence of text edits when there is at least one preceding and one succeeding state in which the code is parsable. The algorithm uses these parsable states to "jump over the gap" of intermediate unparsable states, if any. A scenario without a preceding and succeeding parsable state could cause the algorithm to produce some noise in the form of spurious or missing AST node operations. Nevertheless, in this study, only 0.4% of the inferencing algorithm applications on the data were performed when the edited code contained some parsing errors, and thus, we think that the impact of parsing errors on our results is minimal.

Chapter 3

Inferring Refactorings from Continuous Code Changes

3.1 Introduction

Several research projects [34, 77, 80, 98–100, 104, 132, 140] made strides into understanding the practice of refactoring. The fundamental technical problem in understanding the practice is being able to identify the refactorings that were applied by developers. There are a few approaches. One is to bring developers into the lab and watch how they refactor [99]. This has the advantage of observing all code changes, so it is precise. But this approach studies the programmers in a confined environment, for a short period of time, and thus, it is *unrepresentative*.

Another approach is to study the refactorings applied in the wild. The most common way is to analyze two Version Control System (VCS) snapshots of the code either manually [11,34,93,94] or automatically [5, 31,32,52,82,134,137]. However, the snapshot-based analysis has several disadvantages. First, it is *imprecise*. Many times refactorings overlap with editing sessions, e.g., a method is both renamed, and its method body is changed dramatically. Refactorings can also overlap with other refactorings, e.g., a method is both renamed and its arguments are reordered. The more overlap, the more noise. In our study on code evolution (see Section 2.3), 46% of refactored program entities are also edited or further refactored in the same commit. Second, it is *incomplete*. For example, if a method is renamed more than once, a snapshot-based analysis would only infer the last refactoring. Third, it is *impossible* to answer many empirical questions. For example, from snapshots we cannot determine how long it takes developers to refactor, and we cannot compare manual vs. automated refactorings.

Others [100,132] have studied the practice of automated refactorings recorded by Eclipse [34,63], but this approach does not take into account the refactorings that are applied manually. Recent studies [99,100,132] have shown that programmers sometimes perform a refactoring manually, even when the IDE provides an automated refactoring. Thus, this approach is *insufficient*.

To address these five serious limitations, we studied refactoring in the wild, while employing a *continuous* analysis. Such analysis tracks code changes as soon as they happen rather than inferring them from VCS

Scope	Refactoring
API-level	Encapsulate Field Rename Class Rename Field Rename Method
Partially local	Convert Local Variable to Field Extract Constant Extract Method
Completely local	Extract Local Variable Inline Local Variable Rename Local Variable

Table 3.1: Inferred refactorings. *API-level* refactorings operate on the elements of a program's API. *Partially local* refactorings operate on the elements of a method's body, but also affect the program's API. *Completely local* refactorings affect elements in the body of a single method only.

snapshots. Besides allowing researchers to study better the practice of refactoring, inferring refactorings from continuous code changes provides an important benefit to developers — it can be performed on-the-fly and thus, it enables an IDE to assist a developer with refactorings while they are in progress.

Recent tools [44, 49] that were developed for such inference were not designed for empirical studies. Therefore, we designed and implemented our own refactoring inference algorithm that analyzes code changes continuously. Our refactoring inference algorithm takes as input *add*, *delete*, and *update* AST node operations that we infer from the continuous sequence of raw code edits (see Section 2.2). Currently, our algorithm infers ten kinds of refactorings performed either manually or automatically, but it can be easily extended to handle other refactorings as well. These ten kinds of refactorings were previously reported [132] as the most popular among automated refactorings. Table 3.1 shows the inferred refactorings, ranging from API-level refactorings (e.g., Rename Class), to partially local (e.g., Extract Method), to completely local refactorings (e.g., Extract Local Variable). We think the inferred refactorings are representative since they are both popular and cover a wide range of common refactorings that operate on different scope levels. In the following, when we refer to refactorings we mean these ten refactoring kinds.

3.2 Refactoring Inference Algorithm

3.2.1 Inferring Migrated AST Nodes

Many kinds of refactorings that we would like to infer rearrange elements in the refactored program. To correctly infer such refactorings, we need to track how AST nodes migrate in the program's AST. A node might migrate from a single site to another single site (i.e., this node is moved from one parent node to another parent node), for example, as a result of Inline Local Variable refactoring applied to a variable with

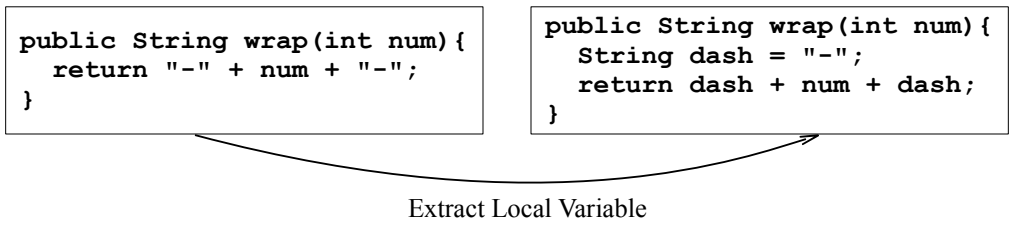


Figure 3.1: An example of Extract Local Variable refactoring that results in many-to-one migration of the extracted AST node.

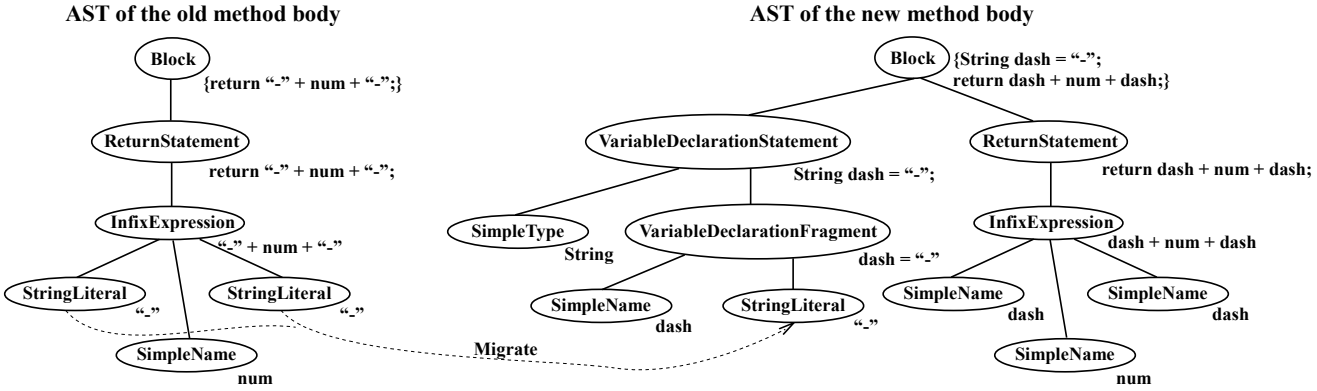


Figure 3.2: The effect of the Extract Local Variable refactoring presented in Figure 3.1 on the underlying AST.

a single usage. Such migration is *one-to-one*. Also, a node might migrate from a single site to multiple sites, e.g., as a result of Inline Local Variable refactoring applied to a variable with multiple usages in the code. Such migration is *one-to-many*. Finally, a node might migrate from multiple sites to a single site, e.g., as a result of Extract Local Variable refactoring applied to an expression that appears in multiple places in the code. Such migration is *many-to-one*.

Figure 3.1 shows an example of Extract Local Variable refactoring that results in many-to-one migration of the extracted AST node. Figure 3.2 shows the effect of this refactoring on the underlying AST. Note that the extracted AST node, string literal `"-"`, is deleted from two places in the old AST and inserted in a single place in the new AST — as the initialization of the newly created local variable.

Our refactoring inference algorithm takes as input a sequence of *basic* AST node operations: *add*, *delete*, and *update*. The algorithm infers *migrate* operations from these basic operations. A single *migrate* operation is composed either of one *delete* operation and one or more *add* or *update* operations, or of one *add* or *update* operation and one or more *delete* operations applied on the same AST node within a specific time window. We consider that two AST nodes represent the same node if they have the same AST node type and the same content. As a time window, we employ a five minutes time interval.

The algorithm assigns a unique ID to each inferred *migrate* operation. Note that a basic AST node

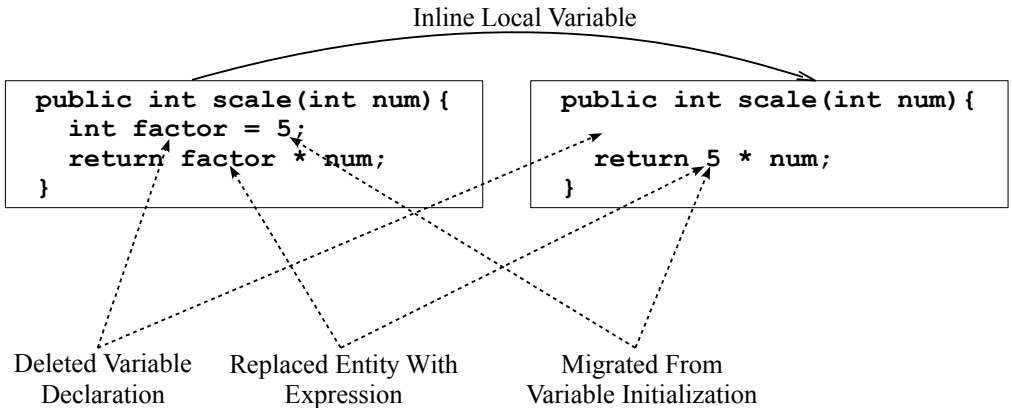


Figure 3.3: An example of Inline Local Variable refactoring and its characteristic properties.

operation can be part of at most one *migrate* operation. The algorithm marks each basic AST node operation that is part of a particular *migrate* operation with its ID. This enables us to easily establish whether two basic AST node operations belong to the same *migrate* operation in the following stages of our refactoring inference algorithm.

3.2.2 Algorithm Overview

Our algorithm infers ten kinds of refactorings shown in Table 3.1. To infer a particular kind of refactoring, our algorithm looks for *properties* that are *characteristic* to it. A refactoring property is a high-level semantic code change, e.g., addition or deletion of a variable declaration. Figure 3.3 shows an example of Inline Local Variable refactoring and its characteristic properties: deletion of a variable declaration, deletion of a variable reference, and migration of the variable's initialization expression to the former usage of the variable.

Refactoring properties are identified directly from the basic AST node operations that represent the actions of a developer. A developer may change the code in any order, e.g., first delete the variable declaration and then replace its references with the initialization expression, or first replace the references and then delete the variable declaration, etc. Consequently, the order in which the properties are identified does not matter.

A refactoring property is described with its *attributes*, whose values are derived from the corresponding AST node operation. Table 3.2 shows 13 attributes that our algorithm employs for a variety of refactoring properties. A property may contain one or more such attributes. Table 3.3 presents refactoring properties and their attributes. When the algorithm checks whether a property can be part of a particular refactoring, the property's attributes are matched against attributes of all other properties that already make part of this refactoring. As a basic rule, two attributes match if either they have different names or they have the same value. Additionally, the algorithm checks that the *disjoint* attributes have different values:

Attribute name	Description
entityName	The name of a program entity
oldEntityName	The old name of a program entity
newEntityName	The new name of a program entity
migratedNode	The migrated AST node
migrateID	The ID of the migrate operation
parentID	The ID of the parent node
destinationMethodID	The ID of the destination method
sourceMethodName	The name of the source method
sourceMethodID	The ID of the source method
getterMethodName	The name of a getter method
getterMethodID	The ID of the gettern method
setterMethodName	The name of the setter method
setterMethodID	The ID of the setter method

Table 3.2: Attributes of refactoring properties.

`destinationMethodID` should be different from `sourceMethodID` and `getterMethodID` should be different from `setterMethodID`.

Our algorithm combines two or more closely related refactoring properties in a single refactoring *fragment*. Such fragments express high-level properties that could not be derived from a single AST node operation, e.g., replacing a reference to an entity with an expression involves two AST node operations: *delete* entity reference and *add* expression. Table 3.4 shows the inferred refactoring fragments and their component properties.

The algorithm considers that a refactoring is *complete* if all its *required* characteristic properties or fragments are identified within a specific time window (in our experiments, we employed a time window of five minutes). Some characteristic properties are optional, e.g., replacing field references with getters and setters in Encapsulate Field refactoring is optional. Also, a refactoring might include several instances of the same characteristic property. For example, an Inline Local Variable refactoring applied to a variable that is used in multiple places includes several properties of migration of the variable's initialization expression to the former usage of the variable. Even though it is sufficient to have a single instance of each required characteristic property to infer a refactoring, our algorithm infers a refactoring as fully as possible, incorporating all properties that belong to it. If no more properties are added to a complete refactoring within two minutes, the algorithm considers that the inference of this refactoring is finished. Table 3.5 presents the characteristic properties of the ten refactorings inferred by our algorithm.

Putting It All Together. Tables 3.2 – 3.5 are populated according to the grammar of our simple refactoring matching language, which we define as follows:

Property name	Property attributes
Added Entity Reference	entityName parentID
Added Field Assignment	entityName setterMethodID
Added Field Declaration	entityName
Added Field Return	entityName getterMethodID
Added Getter Method Declaration	getterMethodName getterMethodID
Added Getter Method Invocation	getterMethodName parentID
Added Method Declaration	entityName destinationMethodID
Added Method Invocation	entityName sourceMethodName sourceMethodID
Added Setter Method Declaration	setterMethodName setterMethodID
Added Setter Method Invocation	setterMethodName parentID
Added Variable Declaration	entityName
Changed Entity Name In Usage	oldEntityName newEntityName sourceMethodName
Changed Field Name In Declaration	oldEntityName newEntityName
Changed Method Name In Declaration	oldEntityName newEntityName
Changed Type Name In Constructor	oldEntityName newEntityName
Changed Type Name In Declaration	oldEntityName newEntityName
Changed Variable Name In Declaration	oldEntityName newEntityName sourceMethodName
Deleted Entity Reference	entityName parentID
Deleted Variable Declaration	entityName
Made Field Private	entityName
Migrated From Method	sourceMethodID migrateID
Migrated From Usage	migratedNode migrateID parentID
Migrated From Variable Initialization	entityName migratedNode migrateID
Migrated To Field Initialization	entityName migratedNode migrateID
Migrated To Method	entityName destinationMethodID migrateID
Migrated To Usage	migratedNode migrateID parentID
Migrated To Variable Initialization	entityName migratedNode migrateID

Table 3.3: Refactoring properties.

Fragment name	Component properties
Migrated Across Methods	Migrated From Method Migrated To Method
Replaced Entity With Expression	Migrated To Usage Deleted Entity Reference
Replaced Entity With Getter	Added Getter Method Invocation Deleted Entity Reference
Replaced Entity With Setter	Added Setter Method Invocation Deleted Entity Reference
Replaced Expression With Entity	Migrated From Usage Added Entity Reference

Table 3.4: Refactoring fragments.

Refactoring	Properties/Fragments	Optional	Multiple instances
Convert Local Variable to Field	Added Field Declaration	no	no
	Deleted Variable Declaration	no	no
Encapsulate Field	Added Getter Method Declaration	no	no
	Added Setter Method Declaration	no	no
	Added Field Assignment	no	no
	Added Field Return	no	no
	Made Field Private	no	no
	Replaced Entity With Getter	yes	yes
	Replaced Entity With Setter	yes	yes
Extract Constant	Added Field Declaration	no	no
	Migrated To Field Initialization	no	no
	Replaced Expression With Entity	no	yes
Extract Local Variable	Added Variable Declaration	no	no
	Migrated To Variable Initialization	no	no
	Replaced Expression With Entity	no	yes
Extract Method	Added Method Declaration	no	no
	Added Method Invocation	no	no
	Migrated Across Methods	no	yes
Inline Local Variable	Deleted Variable Declaration	no	no
	Migrated From Variable Initialization	no	no
	Replaced Entity With Expression	no	yes
Rename Class	Changed Entity Name In Usage	yes*	yes
	Changed Type Name In Constructor	yes*	yes
	Changed Type Name In Declaration	no	no
Rename Field	Changed Entity Name In Usage	no	yes
	Changed Field Name In Declaration	no	no
Rename Local Variable	Changed Entity Name In Usage	no	yes
	Changed Variable Name In Declaration	no	no
Rename Method	Changed Entity Name In Usage	no	yes
	Changed Method Name In Declaration	no	no

Table 3.5: Characteristic properties of the inferred refactorings. Note that at least one of the two optional properties of the *Rename Class* refactoring, *Changed Entity Name In Usage* and *Changed Type Name In Constructor*, is required for this refactoring to be considered complete.

<*name*> ::= LITERAL-STRING

<*value*> ::= LITERAL-STRING

<*attribute*> ::= (<*name*>, <*value*>)

<*property*> ::= <*property*> <*attribute*> | <*attribute*>

<*fragment*> ::= <*fragment*> <*property*> | <*property*>

<*refactoring*> ::= <*refactoring*> <*fragment*> | <*refactoring*> <*property*> | <*fragment*> | <*property*>

To decide whether a property `P1` can be part of a refactoring `R`, we employ the following matching rules:

1. `P1` is characteristic of `R` according to Table 3.5.
2. `R` does not contain a property `P2` of the same kind (according to Table 3.3) as `P1` or `R` accepts multiple properties of the `P1` kind according to Table 3.5.
3. `R` does not contain a property `P2` such that there is an attribute `A1` in `P1` and an attribute `A2` in `P2`, and `A1` and `A2` have the same name, but different values.

Matching rules for fragments in a refactoring and properties in a fragment are similar.

Figure 3.4 shows a high-level overview of our refactoring inference algorithm. The algorithm takes as input the sequence of basic AST node operations marked with migrate IDs — *astNodeOperations*. The output of the algorithm is a sequence of the inferred refactoring — *inferredRefactorings*. The algorithm assigns a unique ID to each inferred refactoring and marks all basic AST node operations that contribute to a refactoring with the refactoring's ID.

The refactoring inference algorithm processes each basic AST node operation from *astNodeOperations* (lines 4 – 17). First, the algorithm removes old pending complete refactorings from *pendingCompleteRefactorings* and adds them to *inferredRefactorings* (line 5). A complete refactoring is considered old if no more properties were added to it within two minutes. Also, the algorithm removes timed out pending incomplete refactorings from *pendingIncompleteRefactorings* (line 6) as well as timed out pending refactoring fragments from *pendingRefactoringFragments* (line 7). An incomplete refactoring or a refactoring fragment times out if it was created more than five minutes ago, i.e., the algorithm allocates a five minutes time window for a refactoring or a refactoring fragment to become complete.

Next, the algorithm generates refactoring properties specific to a particular AST node operation (line 8). The kind of the AST node operation (*add*, *delete*, or *update*), the type of the affected node (e.g., a variable declaration or reference, a method declaration, etc.), the context of the affected node (e.g., the containing method, the containing field or variable declaration, etc.), whether this operation is part of a

```
input: astNodeOperations // the sequence of basic AST node operations
output: inferredRefactorings
1  inferredRefactoringKinds = getAllInferredRefactoringKinds();
2  inferredRefactorings = ∅; pendingCompleteRefactorings = ∅;
3  pendingIncompleteRefactorings = ∅; pendingRefactoringFragments = ∅;
4  foreach (astNodeOperation ∈ astNodeOperations) {
5    inferredRefactorings ∪= removeOldRefactorings(pendingCompleteRefactorings);
6    removeTimedOutRefactorings(pendingIncompleteRefactorings);
7    removeTimedOutRefactoringFragments(pendingRefactoringFragments);
8    newProperties = getProperties(astNodeOperation);
9    foreach (newProperty ∈ newProperties) {
10     handleRefactoringFragments(newProperty, newProperties, pendingRefactoringFragments);
11     isPropertyConsumed = handleCompleteRefactorings(newProperty, pendingCompleteRefactorings);
12     if (isPropertyConsumed) continue;
13     isPropertyConsumed = handleIncompleteRefactorings(newProperty, pendingIncompleteRefactorings,
                                                         pendingCompleteRefactorings);
14     if (isPropertyConsumed) continue;
15     createNewRefactorings(newProperty, inferredRefactoringKinds, pendingIncompleteRefactorings);
16   }
17 }
18 inferredRefactorings ∪= pendingCompleteRefactorings;
```

Figure 3.4: Overview of our refactoring inference algorithm.

migrate operation — all are the factors that the algorithm accounts for in order to generate one or more properties shown in Table 3.3.

In the following step, the algorithm processes the generated properties one by one (lines 9 – 16). First, every new property is checked against pending refactoring fragments (line 10). Next, the algorithm tries to add the new property to pending complete refactorings (line 11). If the new property is added to a complete refactoring, it is consumed and the algorithm proceeds to the next new property (line 12). If the property is not consumed by a complete refactoring, the algorithm checks whether this property can be added to pending incomplete refactorings (line 13). If adding the new property to a pending incomplete refactoring makes it complete, the property is consumed, and the algorithm proceeds to the next new property (line 14). Otherwise, the algorithm creates new refactorings of the kinds that the new property is characteristic of (line 15).

Finally, after processing all AST node operations, the algorithm adds to *inferredRefactorings* any of the remaining pending complete refactorings (line 18).

Figure 3.5 shows the details of checking a new property against pending refactoring fragments. If there is a refactoring fragment that accepts the new property and becomes complete, then this refactoring fragment itself turns into a new property to be considered by the algorithm (line 6). Note that as discussed above, a refactoring fragment or a pending refactoring accepts a property if the property's attributes match the

```
procedure: handleRefactoringFragments(newProperty, newProperties, pendingRefactoringFragments)
1  foreach (pendingRefactoringFragment ∈ pendingRefactoringFragments) {
2    if (accepts(pendingRefactoringFragment, newProperty) {
3      addProperty(pendingRefactoringFragment, newProperty);
4      if (isComplete(pendingRefactoringFragment) {
5        remove(pendingRefactoringFragments, pendingRefactoringFragment);
6        newProperties ∪= pendingRefactoringFragment; break;
7      }
8    }
9  }
10 if (canBePartOfRefactoringFragment(newProperty) {
11   pendingRefactoringFragments ∪= createRefactoringFragment(newProperty);
12 }
```

Figure 3.5: This procedure checks the new property against pending refactoring fragments.

```
function: handleCompleteRefactorings(newProperty, pendingCompleteRefactorings)
1  foreach (pendingCompleteRefactoring ∈ pendingCompleteRefactorings) {
2    if (accepts(pendingCompleteRefactoring, newProperty) {
3      addProperty(pendingCompleteRefactoring, newProperty);
4      return true; // the property is consumed
5    }
6  }
7  return false; // the property is not consumed
```

Figure 3.6: This function tries to add the new property to pending complete refactorings. The function returns `true` if the property is added to a refactoring and `false` otherwise.

attributes of the properties that already make part of the fragment or the refactoring. If the new property can be part of a new refactoring fragment, the algorithm creates the fragment and adds it to *pendingRefactoringFragments* (lines 10 – 12).

Figure 3.6 shows how the algorithm tries to add the new property to pending complete refactorings. If the new property is added to a complete refactoring, the property is consumed, and thus, the function returns `true` (line 4). If there is no complete refactoring that consumes the new property, the function returns `false` (line 7).

Figure 3.7 shows how the algorithm checks whether the new property can be added to pending incomplete refactorings. If an incomplete refactoring accepts the property, it is added to a copy of this incomplete refactoring (lines 3 – 4). This ensures that the initial incomplete refactoring remains unchanged in *pendingIncompleteRefactorings* and thus, could be considered for future properties, if there are any. If adding the new property makes the new refactoring complete, it is added to *pendingCompleteRefactorings* (line 6) and the function returns `true` since the property is consumed (line 7). Otherwise, the new refactoring is added to *pendingIncompleteRefactorings* (line 8). If the new property does not make any of the pending incomplete

```
function: handleIncompleteRefactorings(newProperty,pendingIncompleteRefactorings,
                                       pendingCompleteRefactorings)
1  foreach (pendingIncompleteRefactoring ∈ pendingIncompleteRefactorings) {
2    if (accepts(pendingIncompleteRefactoring, newProperty) {
3      newRefactoring = clone(pendingIncompleteRefactoring);
4      addProperty(newRefactoring, newProperty);
5      if (isComplete(newRefactoring) {
6        pendingCompleteRefactorings ∪= newRefactoring;
7        return true; // the property is consumed
8      } else pendingIncompleteRefactorings ∪= newRefactoring;
9    }
10 }
11 return false; // the property is not consumed
```

Figure 3.7: This function tries to add the new property to pending incomplete refactorings. The function returns `true` if adding the property to an incomplete refactoring makes it complete and `false` otherwise.

```
procedure: createNewRefactorings(newProperty,inferredRefactoringKinds,pendingIncompleteRefactorings)
1  foreach (inferredRefactoringKind ∈ inferredRefactoringKinds) {
2    if (isCharacteristicOf(inferredRefactoringKind, newProperty) {
3      newRefactoring = createRefactoring(inferredRefactoringKind, newProperty);
4      pendingIncompleteRefactorings ∪= newRefactoring;
5    }
6  }
```

Figure 3.8: This procedure creates new refactorings of the kinds that the new property is characteristic of.

refactorings complete (and thus, the property is not consumed), the function returns `false` (line 11).

Figure 3.8 shows how the algorithm creates new refactorings of the kinds that the new property is characteristic of (line 3) and adds these new refactorings to *pendingIncompleteRefactorings* (line 4).

3.2.3 Algorithm Evaluation

Unlike the authors of the other two similar tools [44,49], we report the accuracy of our continuous refactoring inference algorithm on *real world* data. First, we evaluated our algorithm on the automated refactorings performed by the participants of our empirical study (see Section 3.3), which are recorded precisely by Eclipse. We considered 2,398 automated refactorings of nine out of the ten kinds that our algorithm infers (we disabled the inference of the automated Encapsulate Field refactoring in our experiment because the inferencer did not scale for one participant, who performed many such refactorings one after another). A challenge of any inference tool is to establish the *ground truth*, and we are the first to use such a large ground truth. Our algorithm correctly inferred 99.3% of these 2,398 refactorings. The uninferred 16 refactorings represent unlikely code editing scenarios, e.g., ten of them are the Extract Local Variable refactorings in which Eclipse re-writes huge chunks of code in a single shot.

Refactoring	True positives	False negatives	False positives
Convert Local Variable to Field	1	0	1
Encapsulate Field	0	0	0
Extract Constant	0	0	0
Extract Local Variable	8	0	0
Extract Method	2	0	1
Inline Local Variable	2	0	0
Rename Class	3	0	0
Rename Field	5	0	0
Rename Local Variable	28	0	2
Rename Method	4	0	0
Total	53	0	4

Table 3.6: Sampling results.

Also, we randomly sampled 16.5 hours of code development from our corpus of 1,520 hours. Each sample is a 30-minute chunk of development activity, which includes writing code, refactoring code, running tests, committing files, etc. To establish the ground truth, we manually replayed each sample and recorded any observed refactorings of the ten kinds supported by our algorithm. Then, we compared this to the numbers reported by our inference algorithm. Finally, we classified the observed discrepancies as either false positives or false negatives. Table 3.6 shows the sampling results for each kind of refactoring that our algorithm infers. Overall, our inference algorithm has a precision of 0.93 and a recall of 1.

3.3 A Comparative Study of Manual and Automated Refactorings

3.3.1 Introduction

Our continuous refactoring inference algorithm enabled us to perform the first empirical study that addresses the five serious limitations of the existing approaches (which are described in Section 3.1). We study synergistically the practice of both manual and automated refactorings. We answer seven research questions:

RQ1: What is the proportion of manual vs. automated refactorings?

RQ2: What are the most popular automated and manual refactorings?

RQ3: How often does a developer perform manual vs. automated refactorings?

RQ4: How much time do developers spend on manual vs. automated refactorings?

RQ5: What is the size of manual vs. automated refactorings?

RQ6: How many refactorings are clustered?

RQ7: How many refactorings do not reach VCS?

In our study, we applied the refactoring inference algorithm on the real code evolution data from 23 developers, working in their natural environment for 1,520 hours. We found that more than half of the refactorings were performed manually, and thus, the existing studies that focus on automated refactorings only might not be generalizable since they consider less than half of the total picture. We also found that the popularity of automated and manual refactorings differs. Our results present a fuller picture of the popularity of refactorings in general, which should help both researchers and tool builders to prioritize their work. Our findings provide additional evidence that developers underuse automated refactoring tools, which raises the concern of the usability problems in these tools. We discovered that more than one third of the refactorings performed by developers are clustered. This result emphasizes the importance of researching refactoring clusters in order to identify refactoring composition patterns. Finally, we found that 30% of the performed refactorings do not reach the VCS. Thus, using VCS snapshots alone to analyze refactorings might produce misleading results.

3.3.2 Research Methodology

To answer our research questions, we employed the code evolution data that we collected as part of our previous user study (see Section 2.3) on 23 participants. We recruited ten professional programmers who worked on different projects in domains such as marketing, banking, business process management, and database management. We also recruited 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at Urbana-Champaign.

The participants of our study have different affiliations and programming experience, and used our tool, CodingTracker, for different amounts of time. Consequently, the total *aggregated* data is non-homogeneous. To see whether this non-homogeneity affects our results, we divided our participants into seven groups along the three above mentioned categories (i.e., affiliation, programming experience, and tool usage time). Table 3.7 shows the detailed statistics for each group as well as for the aggregated data. For every research question, we first present the aggregated result, and then discuss any discrepancies between the aggregated and the group results.

At the time of the study, when CodingTracker recorded the data, we did not have a refactoring inference

Metric	Group							
	Aggregated	Affiliation		Tool usage (hours)		Programming experience (years)		
		Students	Professionals	≤ 50	> 50	< 5	$5 - 10$	> 10
Participants	23	13	10	13	10	5	11	6
Usage time, hours	1,520	1,048	471	367	1,152	269	775	458
Mean	66	81	47	28	115	54	70	76
STDEV	52	54	44	16	38	46	52	62

Table 3.7: Size and usage time statistics of the aggregated and individual groups.

algorithm. However, CodingTracker can accurately replay all the code editing events, thus recreating an exact replica of the development session. We replayed the coding sessions and this time, we applied our newly-developed refactoring inference algorithm.

We first applied our AST node operations inference algorithm (see Section 2.2) on the collected raw data to represent code changes as *add*, *delete*, and *update* operations on the underlying AST. These basic AST node operations serve as input to our refactoring inference algorithm.

Next, we answer every research question by processing the output of the refactoring inference algorithm with the question-specific analyzer. Note that our analyzers for **RQ1** – **RQ5** ignore *trivial* refactorings. We consider a refactoring trivial if it affects a single line of code, e.g., renaming a variable with no uses.

3.3.3 Results

RQ1: What is the proportion of manual vs. automated refactorings?

Previous research on refactoring practice either predominantly focused on automated refactorings [98, 100, 132] or did not discriminate manual and automated refactorings [34, 140]. Answering the question about the relative proportion of manual and automated refactorings will allow us to estimate how *representative* automated refactorings are of the total number of refactorings, and consequently, how general are the conclusions based on studying automated refactorings only.

For each of the ten refactoring kinds inferred by our algorithm, we counted how many refactorings were applied using Eclipse automated refactoring tools and how many of the inferred refactorings were applied manually. Fig. 3.9 shows our aggregated results. The last column represents the combined result for all the ten refactoring kinds.

Overall, our participants performed around 11% more manual than automated refactorings (2,820 vs. 2,551). Thus, research focusing on automated refactorings considers less than half of the total picture. Moreover, half of the refactoring kinds that we investigated, Convert Local Variable to Field, Extract Method,

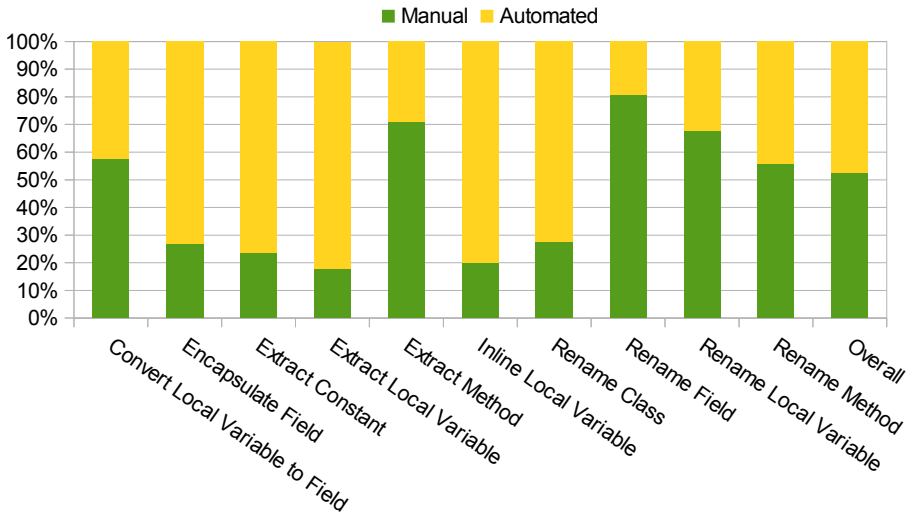


Figure 3.9: Relative proportion of manual and automated refactorings.

Rename Field, Rename Local Variable, and Rename Method, are predominantly performed manually. This observation undermines the generalizability of the existing studies based on the automated execution of these popular refactorings. Also, it raises concerns for tool builders about the underuse of the automated refactoring tools, which could be a sign that these tools require considerable improvement.

We compared the number of manual and automated refactorings performed by each group. Table 3.8 shows the total counts of manual and automated refactorings as well as the relative fraction of manual over automated refactorings. The results for the groups in the **Affiliation** and **Usage time** categories are consistent with the results for the aggregated data. At the same time, the programming experience of our participants has a greater impact on the ratio of the performed manual and automated refactorings. In particular, developers with less than five years of programming experience tend to perform 28% more manual than automated refactorings, while those with an average experience (5 – 10 years) perform more automated than manual refactorings. This result reflects a common intuition that novice developers are less familiar with the refactoring tools (e.g., in **RQ3** we observed that novices do not perform three kinds of automated refactorings at all), but start using them more often as their experience grows. Nevertheless, developers with more than ten years of experience perform many more manual than automated refactorings (49%). One of the reasons for this behavior could be that more experienced developers learned to perform refactorings well before the appearance of the refactoring tools. Also, experts might think that they are faster without the refactoring tool. For example, we observed that such developers mostly perform manually *Rename* refactorings, which could be accomplished quickly (but less reliably) using the *Search & Replace* command.

Category	Group	Manual	Automated	Manual over Automated
Aggregated	All data	2820	2551	10.5%
Affiliation	Students	1645	1516	8.5%
	Professionals	1175	1035	13.5%
Usage time	$\leq$ 50 hours	485	471	3%
	$>$ 50 hours	2335	2080	12.3%
Experience	$<$ 5 years	292	228	28%
	5 – 10 years	1282	1459	-12.1%
	$>$ 10 years	1237	829	49.2%

Table 3.8: Manual and automated refactorings performed by each group.

RQ2: What are the most popular automated and manual refactorings?

Prior studies [98,132] identified the most popular automated refactorings to better understand how developers refactor their code. We provide a more complete picture of the refactoring popularity by looking at both manual and automated refactorings. Additionally, we would like to contrast how similar or different are popularities of automated refactorings, manual refactorings, and refactorings in general.

To measure the popularity of refactorings, we employ the same refactoring counts that we used to answer the previous research question. Fig. 3.10, 3.11, and 3.12 correspondingly show the popularity of automated, manual, and all refactorings in the aggregated data. The Y axis represents refactoring counts. The X axis shows refactorings ordered from the highest popularity rank at the left to the lowest rank at the right.

Our results on popularity of automated refactorings mostly corroborate previous findings [132][1]. The only exceptions are the Inline Local Variable refactoring, whose popularity has increased from the seventh to the third position, and the Encapsulate Field refactoring, whose popularity has declined from the fifth to the seventh position. Overall, our results show that the popularity of automated and manual refactorings is quite different: the top five most popular automated and manual refactorings have only three refactorings in common — Rename Local Variable, Rename Method, and Extract Local Variable, and even these refactorings have different ranks. The most important observation though is that the popularity of automated refactorings does not reflect well the popularity of refactorings in general. In particular, the top five most popular refactorings and automated refactorings share only three refactorings, out of which only one, Rename Method, has the same rank.

Having a fuller picture about the popularity of refactorings, researchers would be able to automate or infer the refactorings that are popular when considering both automated and manual refactorings. Similarly, tool builders should pay more attention to the support of the popular refactorings. Finally, novice developers

[1]Note that we cannot directly compare our results with the findings of Murphy et al. [98] since their data represents related refactoring kinds as a single category (e.g., Rename, Extract, Inline, etc.). For a fair comparison, we would need to aggregate our data using the same categories, but this is not feasible, since we do not infer all refactoring kinds that are part of each such category.

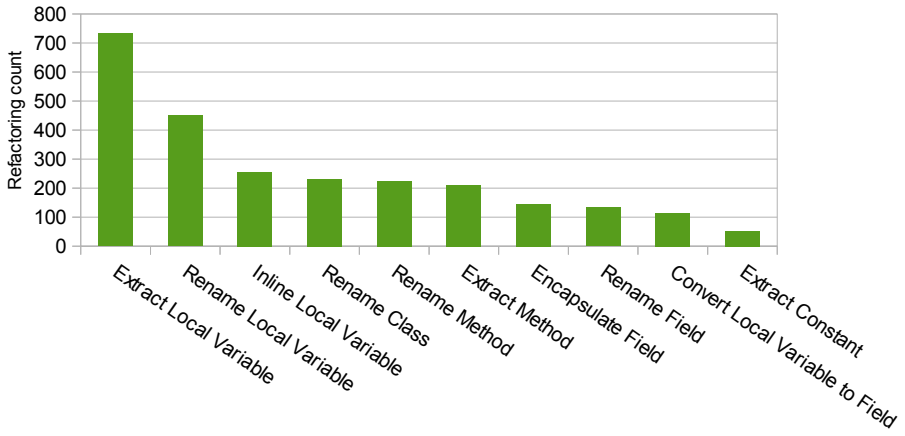


Figure 3.10: Popularity of automated refactorings.

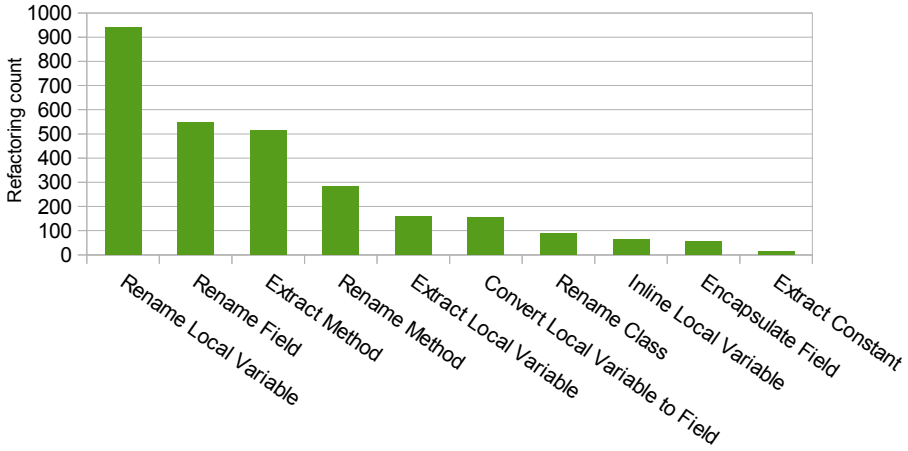


Figure 3.11: Popularity of manual refactorings.

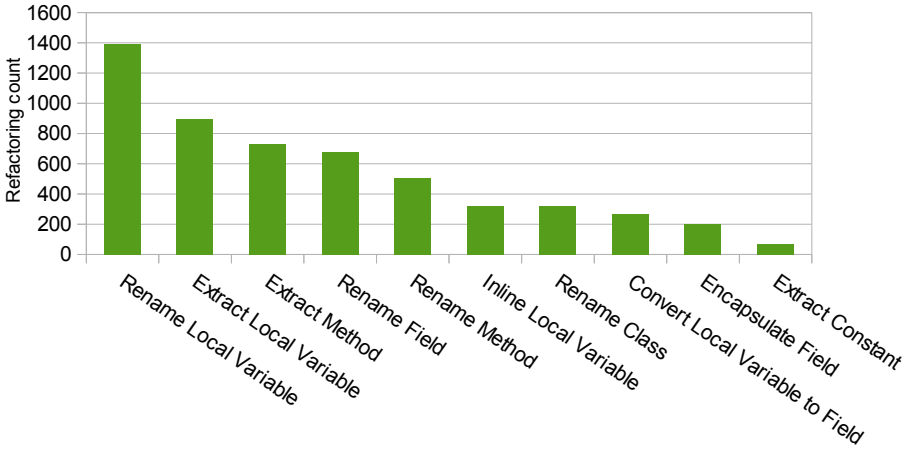


Figure 3.12: Popularity of refactorings.

might decide what refactorings to learn first depending on their relative popularity.

Popularity of refactorings among different participant groups is mostly consistent with the one observed in the aggregated data. In particular, for three groups (Usage time $>$ 50 hours, Experience 5 – 10 years, and Experience $>$ 10 years), the top five most popular refactorings are the same as for the aggregated data, while for the rest of the groups, four out of five most popular refactorings are the same.

RQ3: How often does a developer perform manual vs. automated refactorings?

In our previous study [132], we showed that developers may underuse automated refactoring tools for a variety of reasons, one of the most important being that developers are simply unaware of automated refactoring tools. Answering this question will help us to better understand whether developers who are aware of an automated refactoring tool use the tool rather than refactor manually.

In the following, we denote the quantity of automated tool usage as A. We compute A as a ratio of automated refactorings to the total number of refactorings of a particular kind performed by an individual participant. For each of the ten inferred refactoring kinds, we counted the number of participants for a range of values of A, from $A = 0\%$ (those who never use an automated refactoring tool) up to $A = 100\%$ (those who always use the automated refactoring tool).

Both for the aggregated data and for all groups, we observed that the fraction of participants who always perform a refactoring manually is relatively high for all the ten refactoring kinds (with a few exceptions). Also, in the aggregated data, there are no participants who apply Convert Local Variable to Field, Encapsulate Field, Extract Method, and Rename Field using the automated refactoring tools only. In groups, there are even more refactoring kinds that are never applied using automated refactoring tools only. Overall, our results provide a stronger quantitative support for the previously reported findings [100, 132] that the automated refactoring tools are underused.

To get a better insight into the practice of manual vs. automated refactoring of our participants, we defined three properties:

- **High full automation:** The number of participants who always perform the automated refactoring ($A = 100\%$) is higher than the number of participants who always perform this refactoring manually ($A = 0\%$).
- **High informed underuse:** The number of participants who are aware of the automated refactoring, but still apply it manually most of the time ($0\% < A \leq 50\%$) is higher than the number of participants who apply this refactoring automatically most of the time ($50\% < A \leq 100\%$).

Category	Group	Property		
		High full automation	**High informed underuse**	**General informed underuse**
Aggregated		Extract Constant Rename Class	Extract Method Rename Local Variable	All
Affiliation	Students	-Extract Constant -Rename Class	+Conv. Loc. Var. to Field +Extract Constant +Rename Method	
	Professionals	+Rename Method	-Rename Local Variable	-Extract Constant -Rename Class -Rename Method
Usage time	$\leq$ 50 hours	-Extract Constant +Rename Method		-Extract Constant -Rename Class
	$>$ 50 hours	-Rename Class	+Rename Method	-Extract Constant
Experience	$<$ 5 years	-Extract Constant -Rename Class		-Conv. Loc. Var. to Field -Extract Constant -Inline Local Variable
	5 – 10 years	-Extract Constant	-Rename Local Variable +Conv. Loc. Var. to Field +Extract Constant	-Extract Constant
	$>$ 10 years	-Extract Constant	+Encapsulate Field +Rename Method	

Table 3.9: Manual vs. automated refactoring practice.

- **General informed underuse:** The number of participants who apply the automated refactoring only ($A = 100\%$) is significantly lower than the number of participants who both apply the automated refactoring and refactor manually ($0\% < A < 100\%$).

Table 3.9 shows refactoring kinds that satisfy the above properties for each group as well as for the aggregated data. For each group, we present only the difference with the aggregated result, where "−" marks those refactoring kinds that are present in the aggregated result, but are absent in the group result, and "+" is used in the vice-versa scenario.

Our aggregated results show that only for two refactorings, Extract Constant and Rename Class, the number of participants who always perform the automated refactoring is higher than the number of participants who always perform the refactoring manually. Another important observation is that for two refactoring kinds, Extract Method and Rename Local Variable, the number of participants who are aware of the automated refactoring, but still apply it manually most of the time is higher than the number of participants who apply this refactoring automatically most of the time. This shows that some automated refactoring tools are underused even when developers are aware of them and apply them from time to time. Our results for groups show that students tend to underuse more refactoring tools than professionals. Also, developers with more than five years of experience underuse more refactoring tools that they are aware of

than those with less than five years of experience. At the same time, novice developers do not use three refactoring tools at all, i.e., they always perform the Convert Local Variable to Field, Extract Constant, and Inline Local Variable refactorings manually. Thus, novice developers might underuse some refactoring tools due to lack of awareness, an issue identified in a previous study [132].

The aggregated result shows that for each of the ten refactoring kinds, the number of participants who apply the automated refactoring only is significantly lower than the number of participants who both apply the automated refactoring and refactor manually. The result across all groups shows that no less than seven refactoring kinds satisfy this property. These results show that developers underuse automated refactoring tools, some more so than others, which could be an indication of a varying degree of usability problems in these tools.

RQ4: How much time do developers spend on manual vs. automated refactorings?

One of the major arguments in favor of performing a refactoring automatically is that it takes less time than performing this refactoring manually [131]. We would like to assess this time difference as well as compare the average durations of different kinds of refactorings performed manually.

To measure the duration of a manual refactoring, we consider all AST node operations that contribute to it. Our algorithm marks AST node operations that contribute to a particular inferred refactoring with a generated refactoring's ID, which allows us to track each refactoring individually. Note that a developer might intersperse a refactoring with other code changes, e.g., another refactoring, small bug fixes, etc. Therefore, to compute the duration of a manual refactoring, we cannot subtract the timestamp of the first AST node operation that contributes to it from the timestamp of the last contributing AST node operation. Instead, we compute the duration of each contributing AST node operation separately by subtracting the timestamp of the preceding AST node operation (regardless of whether it contributes to the same refactoring or not) from the timestamp of the contributing AST node operation. If the obtained duration is greater than two minutes, we discard it, since it might indicate an interruption in code editing, e.g., a developer might get distracted by a phone call or take a break. Finally, we sum up all the durations of contributing AST node operations to obtain the duration of the corresponding refactoring.

We get the durations of automated refactorings from CodingSpectator [132]. CodingSpectator measures configuration time of a refactoring performed automatically, which is the time that a developer spends in the refactoring's dialog box. Note that the measured time includes neither the time that the developer might need to check the correctness of the performed automated refactoring nor the time that it takes Eclipse to

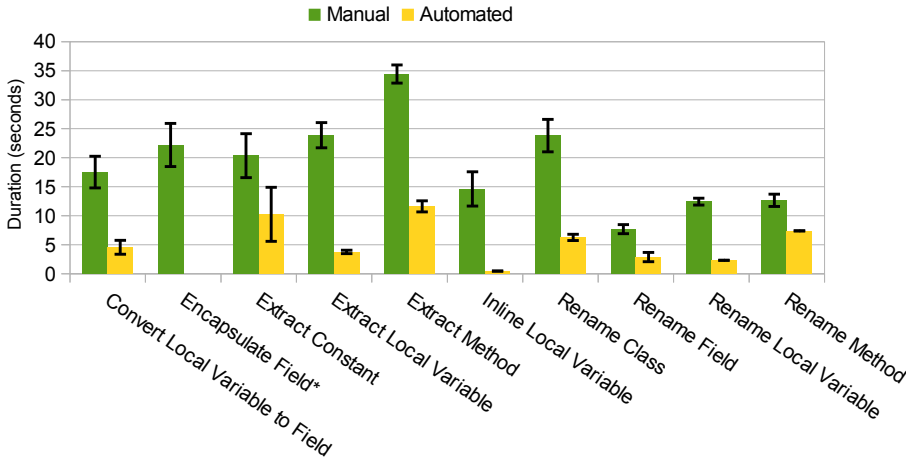


Figure 3.13: Average duration of performing manual refactorings and configuring automated refactorings. The black intervals represent the standard error of the mean (SEM). The configuration time bar for the Encapsulate Field refactoring is missing since we do not have data for it.

actually change the code, which could range from a couple of milliseconds to several seconds, depending on the performed refactoring kind and the underlying code.

Fig. 3.13 shows our aggregated results. On average, manual refactorings take longer than their automated counterparts with a high statistical significance ($p < 0.0001$, using two-sided unpaired t-test) only for Extract Local Variable, Extract Method, Inline Local Variable, and Rename Class since for the other refactoring kinds our participants rarely used the configuration dialog boxes (i.e., our participants mostly performed these refactoring kinds without going through the configuration process). This observation is also statistically significant across all groups. Manual execution of the Convert Local Variable to Field refactoring takes longer than the automated one with a sufficient statistical significance ($p < 0.04$) for the aggregated data, while for most groups this observation is not statistically significant. The most time consuming, both manually and automatically, is the Extract Method refactoring, which probably could be explained by its complexity and the big number of code changes involved. All other refactorings are performed manually on average in under $15 - 25$ seconds. Some refactorings take longer than others. A developer could take into account this difference when deciding what automated refactoring tool to learn first.

Another observation is that the Rename Field refactoring is on average the fastest manual refactoring. It takes less time than the arguably simpler Rename Local Variable refactoring. One of the possible explanations is that developers perform the Rename Field refactoring manually when it does not require many changes, e.g., when there are few references to the renamed field, which is supported by our results for the following question.

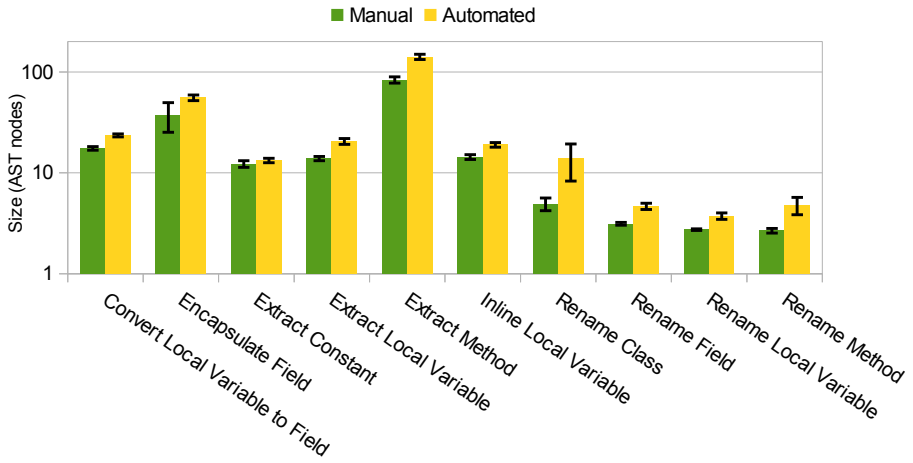


Figure 3.14: Average size of manual and automated refactorings expressed as the number of affected AST nodes. The black intervals represent the standard error of the mean (SEM). The scale of the Y axis is logarithmic.

RQ5: What is the size of manual vs. automated refactorings?

In an earlier project [132], we noticed that developers mostly apply automated refactoring tools for small code changes. Therefore, we would like to compare the average size of manual and automated refactorings to better understand this behavior of developers.

To perform the comparison, we measured the size of manual and automated refactorings as the number of affected AST nodes. For manual refactorings, we counted the number of AST node operations contributing to a particular refactoring. For automated refactorings, we counted all AST node operations that appear between the start and the finish refactoring operations recorded by CodingTracker. Note that all operations between the start and the finish refactoring operations represent the effects of the corresponding automated refactoring on the underlying code [104].

Fig. 3.14 shows our aggregated results. On average, automated refactorings affect more AST nodes than manual refactorings for four refactoring kinds, Convert Local Variable to Field, Extract Method, Rename Field, and Rename Local Variable, with a high statistical significance ($p < 0.0001$), and for three refactoring kinds, Extract Local Variable, Inline Local Variable, and Rename Method, with a sufficient statistical significance ($p < 0.03$). One of the reasons could be that developers tend to perform smaller refactorings manually since such refactorings have a smaller overhead. At the same time, this observation is not statistically significant for all the above seven refactoring kinds in every group. In particular, it is statistically significant in five out of seven groups for three refactoring kinds, Convert Local Variable to Field, Extract Method, and Rename Field, and in fewer groups for the other four kinds of refactorings.

Intuitively, one could think that developers perform small refactorings by hand and large refactorings

with a tool. On the contrary, our findings show that developers perform even large refactorings manually. In particular, Extract Method is by far the largest refactoring performed both manually and automatically – it is more than two times larger than Encapsulate Field, which is the next largest refactoring. At the same time, according to our result for **RQ3**, most of the developers predominantly perform the Extract Method refactoring manually in spite of the significant amount of code change. Thus, the size of a refactoring is not a decisive factor for choosing whether to perform it manually or with a tool. This also serves as an additional indication that the developers might not be satisfied with the existing automation of the Extract Method refactoring [99].

RQ6: How many refactorings are clustered?

To better understand and support refactoring activities of developers, Murphy-Hill et al. [100] identified different refactoring patterns, in particular, *root canal* and *floss* refactorings. A root canal refactoring represents a consecutive sequence of refactorings that are performed as a separate task. Floss refactorings, on the contrary, are interspersed with other coding activities of a developer. In general, grouping several refactorings in a single cluster might be a sign of a higher level refactoring pattern, and thus, it is important to know how many refactorings belong to such clusters.

To detect whether several refactorings belong to the same cluster, we compute a ratio of the number of AST node operations that are part of these refactorings to the number of AST node operations that happen in the same time window as these refactorings, but do not belong to them (such operations could happen either between refactorings or could be interspersed with them). If this ratio is higher than a particular threshold, T, we consider that the refactorings belong to the same cluster. That is, rather than using a specific time window, we try to construct as large clusters as possible, adding refactorings to a cluster as long as the ratio of refactoring to non-refactoring changes in the cluster does not fall below a particular threshold. The minimum size of a cluster is three. Note that for the clustering analysis we consider automated refactorings of all kinds and manual refactorings of the ten kinds inferred by our tool.

Fig. 3.15 shows the proportion of clustered and separate refactorings for aggregated data for different values of T, which we vary from 1 to 10. $T = 1$ means that the number of non-refactoring changes does not exceed the number of refactoring changes in the same cluster. Fig. 3.16 shows the average size of gaps between separate refactorings (i.e., refactorings that do not belong to any cluster) expressed as the number of AST node operations that happen between two separate refactorings or a separate refactoring and a cluster.

Our aggregated results show that for $T = 1$, 45% of the refactorings are clustered. When the threshold grows, the number of the clustered refactorings goes down, but not much — even for $T = 10$, 28% of

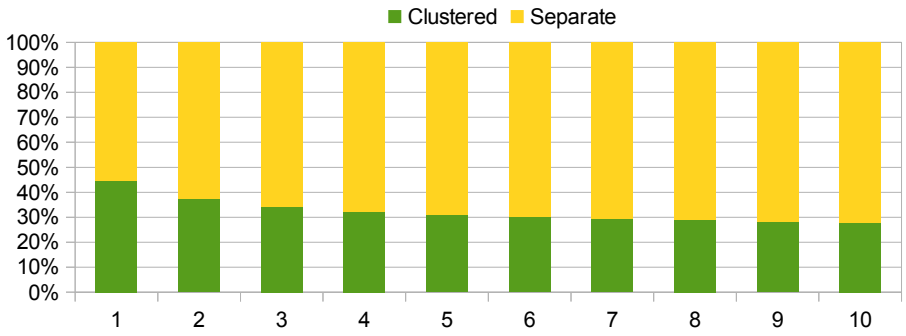


Figure 3.15: Proportion of clustered and separate refactorings for different values of the threshold T.

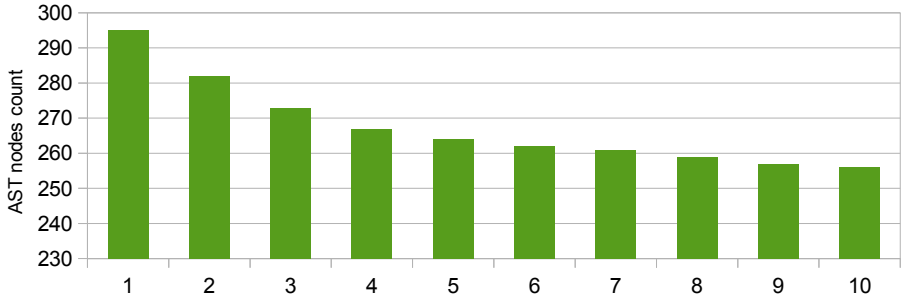


Figure 3.16: The average size of gaps between separate refactorings expressed as the number of AST node operations. The X axis represents the values of the threshold T.

refactorings are clustered. The average gap between floss refactorings is not very sensitive to the value of the threshold as well. Overall, developers tend to perform a significant fraction of refactorings in clusters. This observation holds for all groups except for the novice developers, where for $T = 1$ only 8% of the refactorings are clustered. One of the reasons could be that novices tend to refactor sporadically, while more experienced developers perform refactorings in chunks, probably composing them to accomplish high-level program transformations (e.g., refactor to a design pattern). Our results emphasize the importance of researching refactoring clusters in order to identify refactoring composition patterns.

RQ7: How many refactorings do not reach VCS?

In Section 2.3, we showed that VCS snapshots provide incomplete and imprecise evolution data. In particular, we showed that 37% of code changes do not reach VCS. Since refactorings play an important role in software development, in this study, we would like to assess the number of refactorings that never make it to VCS, and thus, are missed by any analysis based on VCS snapshots. Note that in our study of code evolution, we looked at how much automated refactorings are interspersed with other code changes in the same commit in the aggregated data only, while in this study, we look at both automated and manual refactorings, we

distinguish ten refactoring kinds, we distinguish different groups of participants, and we are able to count individual refactorings that are *completely* missing in VCS (rather than just being partially overlapped with some other changes).

We consider that a refactoring does not reach VCS if none of the AST node operations that are part of this refactoring reach VCS. An AST node operation does not reach VCS if there is another, later operation that affects the same node, up to the moment the file containing this node is committed to VCS. These non-reaching AST node operations and refactorings are essentially *shadowed* by other changes. For example, if a program entity is renamed twice before the code is committed to VCS, the first Rename refactoring is completely shadowed by the second one.

Fig. 3.17 shows the ratio of reaching and shadowed refactorings for the aggregated data. Since even a reaching refactoring might be partially shadowed, we also compute the ratio of reaching and shadowed AST node operations that are part of reaching refactorings, which is shown in Fig. 3.18.

Our aggregated results show that for all refactoring kinds except Inline Local Variable, some fraction of refactorings are shadowed. Overall, 30% of refactorings are *completely* shadowed. The highest shadowing ratio is for the Rename refactorings. In particular, 64% of the Rename Field refactorings do not reach VCS. Thus, using VCS snapshots to analyze these refactoring kinds might significantly skew the analysis results.

Although we did not expect to see any noticeable difference between manual and automated refactorings, our results show that there are significantly more shadowed manual than automated refactorings for each refactoring kind (except Inline Local Variable, which does not have any shadowed refactorings at all). Overall, 40% of manual and only 16% of automated refactorings are shadowed. This interesting fact requires further research to understand why developers underuse automated refactorings more in code editing scenarios whose changes are unlikely to reach VCS.

Another observation is that even refactorings that reach VCS might be hard to infer from VCS snapshots, since a noticeable fraction of AST node operations that are part of them do not reach VCS. This is particularly characteristic of the Extract refactorings, which have the highest ratio of shadowed AST node operations.

Our results for all groups are consistent with the aggregated results with a few exceptions. In particular, the percentage of completely shadowed refactorings for those participants who used our tool for less than 50 hours, is relatively small — 12%, which could be attributed to the fact that such participants did not have many opportunities to commit their code during the timespan of our study. Another observation is that for novice developers, the fraction of completely shadowed automated refactorings is significantly higher than the fraction of completely shadowed manual refactorings (38% vs. 10%). One of the reasons could be that novices experiment more with automated refactoring tools while learning them (e.g., they might perform an

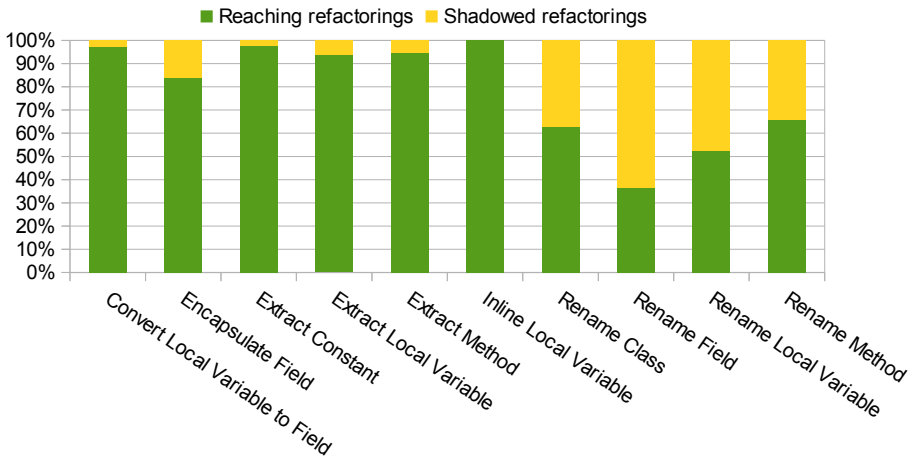


Figure 3.17: Ratio of reaching and shadowed refactorings.

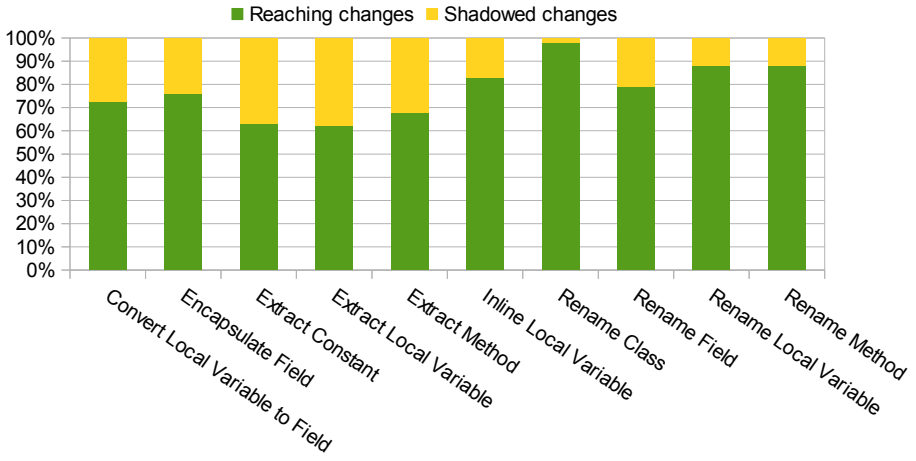


Figure 3.18: Ratio of reaching and shadowed AST node operations that are part of reaching refactorings.

inappropriate automated refactoring and then undo it). Also, novice developers might be less confident in their refactoring capabilities and thus, try to see the outcome of an automated refactoring before deciding how (and whether) to refactor their code, which confirms our previous finding [132]. On the contrary, for the developers with more than ten years of programming experience, the fraction of completely shadowed automated refactorings is very low — 2%, while the fraction of completely shadowed manual refactorings is much higher — 39%. Thus, the most experienced developers tend to perform automated refactorings in code editing scenarios whose changes are likely to reach VCS.

3.3.4 Threats to Validity

Experimental Setup

We encountered difficulties in recruiting a larger group of experienced programmers due to issues such as privacy, confidentiality, and lack of trust in the reliability of research tools. However, we managed to recruit 23 participants, which we consider a sufficiently big group for our kind of study. Our dataset is not publicly available due the non-disclosure agreement with our participants.

Our dataset is non-homogeneous. In particular, our participants have different affiliations and programming experience, and used CodingTracker for a varying amount of time. To address this limitation, we divided our participants into seven groups along these three categories. We answered each research question for every group as well as for the aggregated data and reported the observed discrepancies.

Our results are based on code evolution data obtained from developers who use Eclipse for Java programming. Nevertheless, we expect our results to generalize to similar programming environments.

We infer only ten kinds of refactorings, which is a subset of the total number of refactorings that a developer can apply. To address this limitation to some extent, we inferred those refactoring kinds that are previously reported as being the most popular among automated refactorings [132].

Refactoring Inference Algorithm

Our refactoring inference algorithm takes as input the basic AST node operations that are inferred by another algorithm (see Section 2.2). Thus, any inaccuracies in the AST node operations inference algorithm could lead to imprecisions in the refactoring inference algorithm. However, we compute the precision and recall for both these algorithms applied together, and thus, account for any inaccuracies in the input of the refactoring inference algorithm.

Although the recall of our refactoring inference algorithm is very high, the precision is noticeably lower.

Hence, some of our numbers might be skewed, but we believe that the precision is high enough not to undermine our general observations.

To measure the precision and recall of the refactoring inference algorithm, we sampled around 1% of the total amount of data. Although this is a relatively small fraction of the analyzed data, the sampling was random and involved 33 distinct 30-minute intervals of code development activities, thus a manual analysis of 990 minutes of real code development, which is a significant amount of data for such task.

Chapter 4

Identifying Previously Unknown Program Transformations

4.1 Introduction

Many code changes are repetitive by nature, thus forming code change *patterns*. Frequent pattern mining [59] is successfully applied in a broad range of domains. For example, Amazon.com recommends related products based on "customers who bought this also bought that". Netflix recommends new movies based on "customers who watched this also watched that". Similar frequent pattern mining has revolutionized other services such as ITunes, GoodReads, social platforms, etc. More recently, data mining techniques became popular in the domain of genetics [114, 124, 129]. In particular, these techniques are employed to identify similar sequences of genes, which is a common task in DNA studies. We conjecture that mining frequent code changes can be similarly transformative for software development.

Identifying frequent code change patterns benefits Integrated Development Environment (IDE) designers, code evolution researchers, and developers. IDE designers can build tools that automate execution of frequent code changes, thus improving the productivity of developers. Researchers would better understand the practice of code evolution and also would be able to focus their attention on the most popular development scenarios. Library developers can notice and fix the common mistakes in the library API usage.

Application developers would particularly benefit from the ability of an IDE to identify repetitive code changes *on-the-fly*. An IDE that learns code change patterns as soon as the developer types them can (i) perform the corresponding change automatically the next time a developer needs it or (ii) detect inconsistencies in the code changes if the developer continues to perform them manually. Developers would also benefit if an IDE can learn frequent library usage patterns and offer intelligent code-completion [17, 106, 107] based on most common scenarios. Finally, such an IDE can learn code changes from more experienced programmers and suggest them to novices, thus creating a *virtual* pair-programming environment that does not incur the limitations of forcing both programmers to be collocated in space and time.

Existing research [17, 20, 21, 66, 90, 96, 128, 130, 141, 143, 150] predominantly detects frequent code change patterns either analyzing the static source code of a single version of an application or comparing the appli-

cation's Version Control System (VCS) snapshots. In our empirical study of code evolution (see Section 2.3), we showed that data stored in VCS is *imprecise*, *incomplete*, and makes it *impossible* to perform analysis that involves the time dimension inside a single VCS snapshot. However, the most important limitation both of the snapshot-based techniques and of the static source code analysis is that such approaches cannot learn code changes *on-the-fly*, i.e., while the changes are in progress. Recent research [44, 49] aimed at providing on-the-fly code change assistance to developers, but their code change identification techniques were limited in two ways: (i) they were looking for a single kind of code change patterns — refactorings, (ii) they considered only a small subset of previously known kinds of refactorings.

We employed data mining techniques to detect *previously unknown* frequent code change patterns from a *continuous* sequence of code changes. Since our approach works on a continuous sequence of code changes, it can be applied either on-the-fly, or on a previously recorded sequence.

We designed and implemented a novel frequent code change pattern mining algorithm that accounts for two major challenges specific to the domain of program transformations — overlapping transactions and mining frequent itembags rather than itemsets (we discuss these challenges in Sections 4.2 and 4.4.1). To effectively handle both overlapping transactions and itembags, it is crucial to directly access the transaction identifiers while computing new itemsets. Therefore, our algorithm employs the vertical data format [144]. Our approach is inspired by several ideas from CHARM [145], the state-of-the-art algorithm for frequent closed itemsets mining that uses the vertical data format. In particular, our algorithm extends the notion of itemset-tidset tree (IT-tree) and adapts several optimization insights of CHARM.

We applied our novel algorithm on the real code evolution data from 23 developers, working in their natural environment for 1,520 hours. Our evaluation shows that our algorithm is effective, useful, and scales well for big amounts of data. In particular, our algorithm mined more than half a million item instances in less than six hours. We analyzed some of the frequent code change patterns detected by our algorithm and identified ten kinds of popular high-level program transformations. On average, 32% of the pattern occurrences reported by the algorithm led to high-level program transformation discoveries.

4.2 Motivating Example

Figure 4.1 shows a code editing example in which a developer repeatedly applies a high-level program transformation. In this and all subsequent examples of program transformations, we represent the changed parts of code as underlined text. Figure 4.1 shows that the developer edits two methods, `getDistance` and `computeDirection`. Method `getDistance` computes distance between two points, `p1` and `p2`. Method

Before

```
double getDistance(Point p1, Point p2) {
  ...
}

float computeDirection(Point o, Point p) {
  ...
}
```

After

```
double computeDistance(Point p1, Point p2) {
  if (p1.equals(p2)) return 0;
  ...
}

double computeDirection(Point o, Point p) {
  if (o.equals(p)) return 0;
  ...
}
```

Figure 4.1: An example of a repeated high-level program transformation.

`computeDirection` computes a direction angle from some origin point `o` to a given point `p`. In this code editing example, the developer first renames `getDistance` to `computeDistance` to better reflect its meaning. Then, the developer decides to increase accuracy of direction angle computation and changes accordingly the return type of `computeDirection` method from `float` to `double`. Finally, the developer improves the performance of both methods — if the method's arguments are equal, the method returns `0` without performing any additional computations. We call the corresponding high-level program transformation Return If Arguments Are Equal.

Table 4.1 shows code changes of the code editing scenario in Figure 4.1. The first column presents the relative order of the changes. Note, however, that the ordering of changes is partial. For example, according to the code editing scenario in Figure 4.1, change 2 happened after change 1. At the same time, the mutual order of changes 3 – 9 is undefined. The second and the third columns correspondingly show the kind of the AST node operation and the affected AST node's type. The last column presents the content of the affected AST node (or the content's change for *change* operations).

Given a sequence of code changes like the one in Table 4.1, we would like to identify repetitive code change patterns that correspond to some high-level program transformations. For example, we would like to detect that code changes 3 – 9 and 10 – 16 represent the same code change pattern — the pattern of high-level program transformation Return If Arguments Are Equal. To achieve this goal, we represent each code change as a 2-tuple `(<operation kind>, <AST node type>)`[1]. A code change pattern is an unordered *bag* of code changes. It is a bag rather than a set, since the same code change might occur several times in a pattern, e.g., `(add, SimpleName)` occurs three times in the pattern of transformation Return If Arguments Are Equal.

To detect repetitive code change patterns, we apply data mining techniques. In data mining terminology, an *item* corresponds to our code change and an *itembag* corresponds to our code change pattern. The number

[1] We ignore the contents of the affected AST nodes to avoid making our code changes too specific. Too specific representation hinders detecting similar changes, e.g., changes 9 and 16 in Table 4.1 would be different, if we considered the affected nodes' contents.

Order index	Operation	AST node type	AST node content change
1	*change*	SimpleName	`getDistance` → `computeDistance`
2	*change*	PrimitiveType	`float` → `double`
3	*add*	SimpleName	`p1`
4	*add*	SimpleName	`equals`
5	*add*	SimpleName	`p2`
6	*add*	MethodInvocation	`p1.equals(p2)`
7	*add*	NumberLiteral	`0`
8	*add*	ReturnStatement	`return 0`
9	*add*	IfStatement	`if (p1.equals(p2)) return 0`
10	*add*	SimpleName	`o`
11	*add*	SimpleName	`equals`
12	*add*	SimpleName	`p`
13	*add*	MethodInvocation	`o.equals(p)`
14	*add*	NumberLiteral	`0`
15	*add*	ReturnStatement	`return 0`
16	*add*	IfStatement	`if (o.equals(p)) return 0`

Table 4.1: Code changes of the code editing scenario in Figure 4.1.

of repetitions of a pattern (itembag) is called *frequency*. Note that our goal is to detect full code change patterns rather than their parts (i.e., we would like to detect a pattern with code changes 3 – 9 rather than 3 – 7 or 5 – 8, or any other subset of code changes). Therefore, we mine *closed* itembags, i.e., itembags that are not part of bigger itembags with the same frequency (we present a more formal definition of closedness in the following section).

Note that off-the-shelf data mining algorithms are designed to mine *itemsets* rather than itembags. Also, they assume that *transactions* are disjoint. For code change mining, a transaction is a window in which an algorithm looks for a pattern. The size of the window determines the maximum size of a pattern that a mining algorithm can detect. To mine the actual code changes, we use a time window, while for presentation purposes, we define the size of a window as the number of code changes. For example, let's consider that our window is of size eight. Then, according to table 4.1, we have two disjoint windows (transactions), the first spans code changes 1 – 8, and the second — changes 9 – 16. As a result, the pattern with code changes 3 – 9 crosses the boundary between windows. Consequently, a mining algorithm would not be able to correctly detect the whole pattern. To avoid this problem, we designed a mining algorithm that uses *overlapping* windows (transactions).

In the following, we introduce the canonical problem of mining frequent closed itemsets. Then, we discuss in a more formal way how our problem of mining frequent code change patterns differs from the canonical one, i.e., how we handle itembags and overlapping transactions. We conclude the chapter with the presentation of our code change pattern mining algorithm and its evaluation results.

Transaction identifier (*tid*)	**Set of items**
1	a, c
2	b, d
3	a, b, d
4	c
5	b, c

Table 4.2: An example of a transaction database.

Itemset X	$t(X)$	$sup(X)$
$\{a\}$	$\{1,3\}$	2
$\{a,b\}$	$\{3\}$	1
$\{b,d\}$	$\{2,3\}$	2
$\{a,b,d\}$	$\{3\}$	1

Table 4.3: Example itemsets from the transaction database in Table 4.2.

4.3 Background

A *transaction* is a tuple $< tid, X >$, where *tid* is a unique transaction identifier and X is a set of items. A *transaction database* D is a set of transactions. Let I be a set of all items and T be a set of all *tid*s that are present in a database D. A non-empty set $X \subseteq I$ is called an *itemset*. Correspondingly, a non-empty set $Y \subseteq T$ is called a *tidset*. An itemset that contains n items is called an n-itemset.

A transaction $< tid, X >$ contains itemset Y if Y is a subset of X. Let $t(X)$ denote the set of *tid*s of all transactions that contain itemset X. In a given transaction database D, the *support* of an itemset X, which we denote as $sup(X)$, is the number of transactions in D that contain X. That is, $sup(X) = |t(X)|$. An itemset X is a *frequent* itemset if $sup(X) \geq min\_sup$, where min_sup is a user-specified minimum support threshold. An itemset X is a *closed* itemset if there exists no proper superset Y such that $sup(X) = sup(Y)$. In contrast to mining frequent itemsets, mining frequent closed itemsets produces a significantly more compact result while preserving the result's completeness [113].

Table 4.2 shows an example of a transaction database that contains four items, $I = \{a, b, c, d\}$, and five transactions, $T = \{1, 2, 3, 4, 5\}$. For the convenience of presentation, when the order of items does not matter, we order them alphabetically. Table 4.3 shows several itemsets from the transaction database in Table 4.2. The second and the third columns of Table 4.3 present the set of the containing transaction identifiers and the support for each itemset.

To accommodate such properties of code change pattern mining as overlapping transactions and itembags, it is crucial to access the transaction identifiers directly while computing new itemsets. The vertical data format grants such access to a mining algorithm. Therefore, our approach is inspired by several ideas from

Item	Tidset
a	$\{1, 3\}$
b	$\{2, 3, 5\}$
c	$\{1, 4, 5\}$
d	$\{2, 3\}$

Table 4.4: The transaction database from Table 4.2 represented in the vertical data format.

CHARM [145], an advanced algorithm for mining data in the vertical data format, which introduces the notion of itemset-tidset tree (IT-tree), employs several optimizations, and searches for closed itemsets, thus considerably reducing the size of the mining result. In particular, our algorithm extends the notion of IT-tree and adapts several optimization insights of CHARM. We now introduce the vertical data format and present the CHARM's definition of IT-tree. We discuss our approach in the next section.

The vertical data format represents a transaction database as a set of tuples $< item, tidset >$, where *tidset* is a set of identifiers of transactions that contain the corresponding *item*. Table 4.4 shows the transaction database from Table 4.2 in the vertical data format. Eclat [144] is the first algorithm for mining data in the vertical data format without candidate generation. The basic idea of the algorithm is to compute $(n + 1)$-itemsets from n-itemsets by intersecting their tidsets. The algorithm starts with frequent 1-itemsets and finishes when no more frequent itemsets can be found. For example, let's consider two 1-itemsets, $\{a\}$ and $\{b\}$, from Table 4.4 (note that for any item x there is a corresponding 1-itemset $\{x\}$). The algorithm computes the tidset of a 2-itemset $\{a, b\}$ by intersecting the tidsets of $\{a\}$ and $\{b\}$: $t(\{a, b\}) = t(\{a\}) \cap t(\{b\}) = \{3\}$. The support of the itemset $\{a, b\}$ is 1: $sup(\{a, b\}) = |t(\{a, b\})| = 1$. If the minimum support threshold is greater than 1, then itemset $\{a, b\}$ is discarded. Otherwise, it is added to the results and consequently, it is considered for computing 3-itemsets.

The nodes in an IT-tree are pairs *itemset* : *tidset*. The root of the tree represents an empty itemset, and thus, its tidset is T, the set of all *tid*s. The immediate children of the root node are 1-itemsets that are computed by scanning the transaction database. The immediate children of a non-root node are computed by intersecting this node's tidset with the tidsets of the *not yet considered* 1-itemsets, traversing them from left to right. If the resulting tidset's size falls below the minimum frequency threshold, the new node is not added to the IT-tree. The IT-tree is completed when no more nodes can be added to it. Figure 4.2 shows the IT-tree for the transaction database from Table 4.4. The first level of the tree consists of the 1-itemsets from the database that are paired with their *tid*s, e.g., $\{a\} : \{1, 3\}$, $\{b\} : \{2, 3, 5\}$, etc. The following levels are computed according to the procedure described above. For example, node $\{a, b\} : \{3\}$ is the result of combination of nodes $\{a\} : \{1, 3\}$ and $\{b\} : \{2, 3, 5\}$. Three itemsets in this IT-tree are not closed, and thus, are not included into the mining result: itemsets $\{a, b\}$ and $\{a, d\}$ are subsumed by the proper super itemset

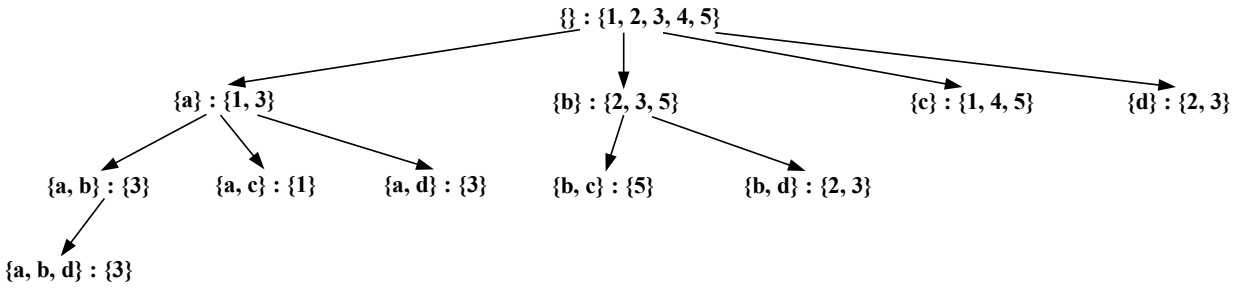


Figure 4.2: The IT-tree for the transaction database from Table 4.4.

$\{a, b, d\}$, while itemset $\{d\}$ is subsumed by the proper super itemset $\{b, d\}$.

4.4 Frequent Code Change Pattern Mining Algorithm

4.4.1 Handling Overlapping Transactions and Itembags

We mine frequent code change patterns from an ordered sequence of code changes (note that code changes are naturally ordered according to when they happened, i.e., according to their timestamps). To populate our transaction database, we divide this continuous sequence into individual transactions. Our code editing example in Section 4.2 shows that making transactions disjoint and sizing them according to the maximum length of a pattern, *max_length*, does not account for patterns that cross the boundary of two transactions. Therefore, we use *overlapping* transactions whose size is $2 * max\_length$. The size of the overlap between two neighboring transactions is *max_length*. As a result, our mining algorithm finds all patterns whose length does not exceed *max_length* and some patterns whose length lies in between *max_length* and $2 * max\_length$.

Figure 4.3 shows an example of a sequence of code changes. For presentation purposes, we denote every distinct kind of code change with a different character. For example, d might stand for `(change, SimpleName)`, a for `(add, MethodInvocation)`, c for `(delete, IfStatement)`, etc. Also, we consider that the maximum length of a pattern we are looking for, *max_length*, is six. For the actual mining, we set *max_length* to five minutes, and thus, transactions contained various numbers of items.

An important observation is that although code changes form an ordered sequence, a code change pattern is unordered because the corresponding high-level program transformation may be performed in different orders. For example, a developer who performs a Rename Local Variable refactoring might first change the variable's declaration and then its references or vice versa, or even intersperse changing the declaration and the references. Thus, the order of a transaction's items does not matter.

Another observation is that a high-level program transformation may contain several instances of the

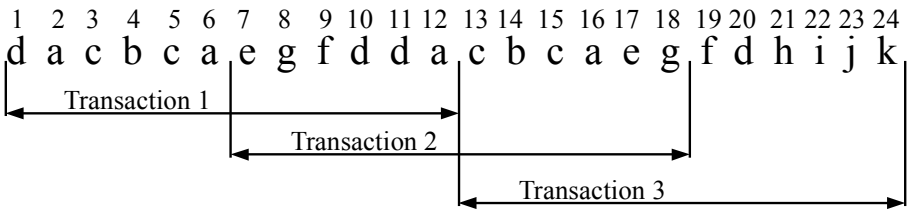


Figure 4.3: An example of a sequence of code changes. Distinct kinds of code changes are represented with different characters.

same kind of code change. For example, Rename Local Variable refactoring involves the same change for all references to the renamed variable. Section 4.2 presents an example of another high-level program transformation, Return If Arguments Are Equal, that also contains multiple instances of the same kind of code change, `(add, SimpleName)`. These examples show that for mining code change patterns, a transaction's items form a bag rather than a set. In particular, the first transaction in Figure 4.3 contains three items a and two items c. Moreover, this transaction contains two instances of code change pattern $\{a, c\}$, which illustrates a scenario when two high-level program transformations happen within max_length time frame.

The major difference between our frequent code change pattern mining algorithm and the existing approaches to mining frequent itemsets is that our algorithm handles overlapping transactions and *itembags* rather than itemsets. To distinguish different *occurrences* of an item in the same transaction as well as the overlapped parts of two transactions, our algorithm assigns a unique ID to each item's occurrence. The first line in Figure 4.3 shows the IDs assigned to the underlying items' occurrences. For example, the first transaction contains occurrences of item a with IDs $\{2, 6, 12\}$, the second transaction — $\{12, 16\}$, and the third — $\{16\}$. Note that although our algorithm handles itembags, we continue to use the notion of itemsets throughout our presentation, since the fact that our itemsets are actually itembags is accounted for by explicitly tracking each item's occurrences.

4.4.2 Tracking an Item's Occurrences

In order to track items' occurrences, a node in our itemset-tidset tree (IT-tree) is defined as follows:

$$
\begin{aligned}
&[item_1, item_2, ..., item_n] : \\
&\quad [tid_1 : [[occurrences_1], [occurrences_2], ..., [occurrences_n]], \\
&\quad\ tid_2 : [[occurrences_1], [occurrences_2], ..., [occurrences_n]], \\
&\quad\ \ ..., \\
&\quad tid_m : [[occurrences_1], [occurrences_2], ..., [occurrences_n]]]
\end{aligned}
$$

Itemset	IT-tree node
$\{a\}$	$[a] : [1 : [[2, 6, 12]], 2 : [[12, 16]], 3 : [[16]]]$
$\{c\}$	$[c] : [1 : [[3, 5]], 2 : [[13, 15]], 3 : [[13, 15]]]$
$\{d\}$	$[d] : [1 : [[1, 10, 11]], 2 : [[10, 11]], 3 : [[20]]]$
$\{a, c\}$	$[a, c] : [1 : [[2, 6, 12], [3, 5]], 2 : [[12, 16], [13, 15]], 3 : [[16], [13, 15]]]$
$\{a, c, d\}$	$[a, c, d] : [1 : [[2, 6, 12], [3, 5], [1, 10, 11]], 2 : [[12, 16], [13, 15], [10, 11]], 3 : [[16], [13, 15], [20]]]$

Table 4.5: Examples of itemsets and their corresponding IT-tree nodes for the sequence of code changes from Figure 4.3.

We use square brackets to denote ordered sets. The order of items in an itemset does not matter for a pattern, but it helps our algorithm to track occurrences of every item in each transaction that contains this itemset. Thus, we represent an n-itemset as an ordered set of items $[item_1, item_2, ..., item_n]$. For a given itemset, a node in an IT-tree contains an ordered set of *tid*s of transactions that contain this itemset. Ordering of transactions enables our algorithm to effectively handle overlapping parts of the neighboring transactions. For each transaction, the IT-tree node also tracks all occurrences for every item in the given itemset (in the above representation, $[occurrences_i]$ are all occurrences of $item_i$ in a particular transaction). Our algorithm also orders an item's occurrences to ensure the optimal result of our itemset frequency computation technique that we discuss below.

Similarly to CHARM [145], we compute our IT-tree by traversing the 1-itemsets from left to right and intersecting the tidset of a particular itemset with the tidsets of the not yet considered 1-itemsets to generate new IT-tree nodes. The major difference from the CHARM's approach is that our algorithm tracks items' occurrences, and thus, whenever a new item is added to an itemset, the item's occurrences are appended to the set of occurrences of every transaction in the corresponding IT-tree node. Table 4.5 shows several examples of itemsets and their corresponding IT-tree nodes for the sequence of code changes from Figure 4.3. The first row presents the IT-tree node for itemset $\{a\}$. The node consists of the itemset itself, $[a]$, followed by *tid*s of transactions that this itemset appears in — $1, 2, 3$. For each transaction, the node tracks all occurrences of item a: transaction 1 contains occurrences $2, 6, 12$, transaction 2 — $12, 16$, and transaction 3 — 16. Note that storing an item's occurrences in every IT-tree node that contains this item is not only redundant, but also prohibitively expensive. Instead, our algorithm stores occurrences of individual items and then just refers these occurrences from the containing IT-tree nodes. We inline the referred occurrences for presentation purposes only.

4.4.3 Computing the Frequency of an Itemset

Due to overlapping transactions and multiple occurrences of an item in the same transaction, our algorithm cannot compute the frequency of an itemset as the number of transactions that contain this itemset (as it is done in the existing frequent itemset mining techniques). Instead, we devised our own itemset frequency computation technique that accounts for the particularities of our mining problem. In a given transaction k:

$tid_k : [[occurrences_1], [occurrences_2], ..., [occurrences_n]]$

the frequency of the corresponding itemset is:

$$f_k = \min_{i=1..n} |[occurrences_i]| \tag{4.1}$$

That is, f_k is the number of occurrences of an itemset's item that appears the least number of times. The *overall* frequency of an itemset that is contained in m transactions is:

$$F = \sum_{k=1}^{m} f_k \tag{4.2}$$

If an item occurrence is shared between two neighboring transactions, k and l, the algorithm should count this occurrence only once, either as part of f_k or as part of f_l. In an ordered set of transactions, two transactions, k and l, are neighboring if and only if $|k-l| = 1$ and $|tid_k - tid_l| = 1$. That is, the neighboring transactions follow each other both in the ordered set and in the original sequence of code changes. For example, in Figure 4.3 transactions of the ordered set $[1, 2]$ are neighboring, while transactions of the ordered set $[1, 3]$ are not neighboring.

Let's denote $[occurrences_i^k]$ the ordered set of occurrences of $item_i$ in a transaction k. Let's denote o_j an occurrence o with the index j in the ordered set of occurrences. To compute the frequency of an n-itemset that is contained in m transactions, our algorithm visits each pair of transactions k and $k+1$, where $1 \leq k < m$. First, our algorithm computes f_k using formula (4.1). Then, if transactions k and $k+1$ are neighboring, our algorithm visits every occurrence $o_j \in [occurrences_i^k]$, where $1 \leq i \leq n$. If $o \in [occurrences_i^{k+1}]$, then our algorithm checks whether the shared occurrence o should be removed from the transaction k or $k+1$. If $j \leq f_k$, then o is removed from the transaction $k+1$. Otherwise, it is removed from the transaction k. Note that removing shared occurrences never affects the initially computed f_k. Finally, our algorithm computes the overall frequency using formula (4.2).

tid	**Occurences**	f_k	f'_k
1	[[2, 6, ~~12~~], [3, 5], [1, 10, ~~11~~]]	2	2
2	[[12, ~~16~~], [13, ~~15~~], [~~10~~, 11]]	1	2
3	[[16], [~~13~~, 15], [20]]	1	1

Table 4.6: The frequency computation result for the itemset $\{a, c, d\}$.

For the best performance of our algorithm, we order an item's occurrences such that those that happened earlier in time appear earlier in the ordered set. Since occurrences' IDs are generated incrementally (see Figure 4.3), such ordering is easily achieved by sorting occurrences in ascending order of their IDs. Consequently, the occurrences that are shared between transactions k and $k+1$ are placed at the end of the ordered set of all occurrences for the transaction k. Hence, our algorithm computes the maximal possible frequency for the transaction k employing the shared occurrences only when needed, while the unused part of them is attributed to the subsequent transaction $k+1$, thus maximizing its frequency too. Going through each pair of transactions k and $k+1$, $1 \leq k < m$, our algorithm propagates this maximization, thus computing the optimal overall frequency F.

Table 4.6 shows the frequency computation result for the itemset $\{a, c, d\}$. Occurrences removed by the algorithm are crossed out. The values in the fourth column, f'_k, are computed by applying the formula in (4.1) without removing the shared occurrences. The overall frequency of the itemset $\{a, c, d\}$ is $F = 4$.

4.4.4 Optimizations

To optimize the mining algorithm, CHARM exploits several insights about the properties of itemsets and tidsets. Let's recall that the computation of immediate children of an IT-tree node of an itemset X involves traversing the not yet considered 1-itemsets from left to right. Let's assume that this traversal visits two 1-itemsets, Y and Z, in this order. If $t(X \cup Y) = t(X \cup Z)$, then there is no need to create separate children nodes corresponding to Y and Z. Instead, CHARM removes both Y and Z from the list of not yet considered 1-itemsets and adds a node $X \cup Y \cup Z$ as a child of the IT-tree node of the itemset X. Similarly, if $t(X \cup Y) \supset t(X \cup Z)$, then a child node $X \cup Y \cup Z$ is added to the IT-tree, but the 1-itemset Z is not removed from the yet not considered 1-itemsets, which means that CHARM would consider Z without Y while computing another immediate child, $X \cup Z$. To maximize the chances of encountering such scenarios, CHARM sorts the not yet considered 1-itemsets in ascending order of the number of transactions that contain them.

In our algorithm, we adapt the CHARM's optimization insights to the specifics of our data mining problem. We introduce the notion of a *frequency descriptor*, FD, a 3-tuple that captures all relevant information

about the frequency of an itemset that is contained in m transactions:

$FD =< F, \vec{f_k}, \vec{f'_k} >$, where $k = 1..m$.

For example, according to Table 4.6, the frequency descriptor of the itemset $\{a, c, d\}$ is $FD =< 4, (2, 1, 1), (2, 2, 1) >$. Two frequency descriptors are equivalent if they have exactly the same elements. A frequency descriptor FD_1 is *more powerful* than a frequency descriptor FD_2 if FD_1 and FD_2 are not equivalent and every element of FD_1 is equal to or greater than the corresponding element of FD_2. For example, $< 4, (2, 1, 1), (2, 2, 1) >$ is more powerful than $< 4, (2, 1, 1), (1, 2, 1) >$, but it is not more powerful than $< 4, (2, 1, 1), (1, 3, 1) >$.

To optimize the computation of the IT-tree, our algorithm first sorts the not yet considered 1-itemsets Y of a given itemset X by the number of transactions that contain $X \cup Y$. If two 1-itemsets, Y and Z, are such that $t(X \cup Y) = t(X \cup Z)$, then the algorithm arranges Y and Z in the lexicographical order of the frequency descriptors of $X \cup Y$ and $X \cup Z$.

Next, our algorithm computes the immediate children of the IT-tree node of the itemset X by traversing the sorted list of 1-itemsets. If two yet not considered 1-itemsets, Y and Z, are such that $t(X \cup Y) = t(X \cup Z)$ and the frequency descriptors of $X \cup Y$ and $X \cup Z$ are equivalent, our algorithm removes both Y and Z from the list of the not yet considered 1-itemsets and adds a node $X \cup Y \cup Z$ as a child of the IT-tree node of the itemset X. If Y and Z are such that $t(X \cup Y) = t(X \cup Z)$ or $t(X \cup Y) \supset t(X \cup Z)$, and the frequency descriptor of $X \cup Z$ is more powerful than the frequency descriptor of $X \cup Y$, our algorithm adds a child node $X \cup Y \cup Z$ to the IT-tree, but keeps the 1-itemset Z in the list of the not yet considered 1-itemsets.

4.4.5 Computing Closed Itemsets

The output of our algorithm contains only closed code change patterns, i.e., for each itemset X that represents a particular pattern, in our result, there is no itemset Y such that $Y \supset X$ and $F_Y = F_X$. To ensure the closedness of the result, our algorithm checks every newly generated itemset X against the already accumulated itemsets. Since the number of accumulated itemsets might be very big, to speed up the checking process, we hash the accumulated itemsets similarly to CHARM, i.e., using the sum of the itemsets' *tids* as the key.

Unlike CHARM though, our algorithm differentiates between *partially subsumed* and *completely subsumed* itemsets as well as detects scenarios, in which the previously accumulated itemset needs to be replaced with the new one. Also, our algorithm compares the frequency descriptor of the newly generated itemset X,

FD_X, with the frequency descriptors of the previously generated itemsets Y, FD_Y. If $Y \supset X$ and FD_Y is more powerful than FD_X, then the itemset X is completely subsumed — it is not added to the results and the algorithm stops computing the children of the corresponding IT-tree node. Otherwise, if $Y \supset X$ and the overall frequency of X, F_X, is equal to the overall frequency of Y, F_Y, then the itemset X is partially subsumed — it is not added to the results, but the algorithm proceeds computing the children of the corresponding IT-tree node. Finally, if $Y \subset X$ and $F_X = F_Y$, Y is removed from the results. Note that all the above decisions are valid only for X and Y that appear in the same transactions, i.e., $t(X) = t(Y)$.

4.4.6 Establishing Frequency Thresholds

To decide whether an itemset is frequent enough to be added to the results, our algorithm employs two thresholds. First, our algorithm checks whether an itemset's overall frequency is not less than the *absolute frequency threshold*, i.e., the frequency threshold that ignores all other characteristics of the itemset (e.g., its size or composition). Next, our algorithm checks whether the itemset's overall frequency multiplied by the itemset's size is not less than the *dynamic threshold*, i.e., the longer an itemset grows, the less frequent it can be in order to pass this threshold. For example, a dynamic threshold of 100 filters out 1-itemsets that are less frequent than 100, 2-itemsets that are less frequent than 50, and so on.

4.4.7 Putting It All Together

Figure 4.4 shows a high-level overview of our frequent code change pattern mining algorithm. The pseudocode of the figure contains all the basic functionality features of our algorithm discussed above.

4.5 Evaluation

In our evaluation, we would like to answer these questions:

- **Q1**(scalability): Is our algorithm scalable to handle big amounts of data?
- **Q2**(effectiveness): Does our algorithm mine code change patterns that simplify identification of high-level program transformations?
- **Q3**(usefulness): Can we detect interesting high-level program transformations from the mined code change patterns?

To answer these questions, we applied our frequent code change pattern mining algorithm on a large corpus of real world data. In the following, we first describe how we collected the data and performed the

```
input: codeChangesSequence
output: frequentPatterns

procedure: mine(codeChangesSequence) {
  frequentPatterns = ∅; emptySet = ∅;
  inputItems = getInputItems(codeChangesSequence);
  solve(emptySet, inputItems);
}

procedure: solve(currentItemset, remainingItems) {
  // Sort remainingItems according to currentItemset as a prefix.
  remainingSortedItems = sort(currentItemset, remainingItems);
  while (remainingSortedItems ≠ ∅) {
    nextItem = pollFirst(remainingSortedItems);
    newItemset = addItem(currentItemset, nextItem);
    if (getFrequency(newItemset) ≥ ABSOLUTE_FREQUENCY) {
      newRemainingItems = copy(remainingSortedItems);
      foreach (forwardItem ∈ remainingSortedItems) {
        freqCompResult = compare(freqDescr(newItemset), freqDescr(addItem(newItemset, forwardItem));
        if (freqCompResult == EQUIVALENT || freqCompResult == SECOND_MORE_POWERFUL) {
          newItemset = addItem(newItemset, forwardItem);
          removeItem(newRemainingItems, forwardItem);
          if (freqCompResult == EQUIVALENT) {
            removeItem(remainingSortedItems, forwardItem);
          }
        }
      }
      if (getFrequency(newItemset) * getSize(newItemset) ≥ DYNAMIC_THRESHOLD) {
        subsumptionResult = checkSubsumption(newItemset);
        if (subsumptionResult ≠ FULLY_SUBSUMED) {
          if (subsumptionResult ≠ PARTIALLY_SUBSUMED) {
            frequentPatterns = addItemset(frequentPatterns, newItemset);
          }
          solve(newItemSet, newRemainingItems);
        }
      }
    }
  }
}
```

Figure 4.4: Overview of our frequent code change pattern mining algorithm.

evaluation of the algorithm. Then, we present our results.

4.5.1 Experimental Setup

We applied our algorithm on the data collected during our previous user study (see Section 2.3), which involved 23 participants: ten professional programmers who worked on different projects in domains such as marketing, banking, business process management, and database management; and 13 Computer Science graduate students and senior undergraduate summer interns who worked on a variety of research projects from six research labs at the University of Illinois at Urbana-Champaign. Table 2.2 shows the programming experience of our participants. We evaluated our mining algorithm on 1,520 hours of code development collected from our participants.

We first applied our AST node operations inference algorithm (see Section 2.2) on the collected raw data to represent code changes as *add*, *delete*, and *update* operations on the underlying AST. Next, we represented distinct kinds of code changes as combinations of the operation and the type of the affected AST node. For example, `(add, IfStatement)`, `(delete, IfStatement)`, and `(add, InfixExpression)` are three different kinds of code changes. The instances of code change kinds serve as input to our frequent code change pattern mining algorithm. That is, in our mining algorithm, a code change kind is an *item* and an instance of a code change kind is an item's *occurrence*.

For each mined code change pattern, our algorithm reports all occurrences of the pattern in the input sequence of code changes. We use CodingTracker's replayer to manually investigate these occurrences. We replay the code changes of a particular occurrence to detect the corresponding high-level program transformation.

Since the mining result is huge, we order the mined patterns along three dimensions: by frequency of the pattern (F), by size of the pattern (S), and by $F * S$. Then, we output the top 1,000 patterns for each dimension and investigate them starting from the top of the list.

Recall that our algorithm uses two thresholds to decide whether a pattern is frequent: absolute frequency threshold and dynamic threshold. Both thresholds check whether a specific pattern's metrics (a pattern's frequency and a pattern's frequency multiplied by its size correspondingly) do not fall below a user-specified value. Providing different values for these thresholds, one can tune the output of our mining algorithm. We noticed that some items (i.e., atomic AST node operations like `(add, IfStatement)`) are much more frequent than the others. Thus, picking a single pair of threshold values to analyze our data is impractical. If these values are too low, our algorithm's scalability would degrade, while the output would become disproportionately big. On the other hand, too high values would hinder the mining of patterns that involve

Frequency, F	NK	AFT	DT	Mining time
$10,000 \leq F$	23	30	10,000	15 minutes
$300 \leq F < 10,000$	81	30	300	5.2 hours
$5 \leq F < 300$	32	5	5	7.7 seconds

Table 4.7: Grouping of item kinds by their frequency. Column **NK** shows the number of item kinds in each group. Columns **AFT** and **DT** show the values of the absolute frequency threshold and dynamic threshold.

Item kinds	Transactions	Item occurrences	Total mining time
136	7,927	549,184	5.5 hours

Table 4.8: Performance statistics of our experiment.

less frequent items. Therefore, we divided the input items into three groups, applying different threshold values to each. Table 4.7 shows each group as well as the corresponding thresholds and the mining time. We performed all mining on a quad-core i7 2GHz machine with 8GB of RAM.

We observed that some AST node operations are too frequent to be considered at all. For example, adding and deleting `SimpleName` accompanies any code change that declares or references a program entity. Consequently, mining items that represent such AST node operations would only add noise to the detected code change patterns. Therefore, before applying our algorithm, we filtered out the noisy item kinds, thus reducing the number of item kinds from 162 to 138. According to Table 4.7, the total number of the considered item kinds is 136, which means that two item kinds were too infrequent to be part of any group.

4.5.2 Results

Table 4.8 summarizes performance statistics of our experiment. Our algorithm mined more than half a million item occurrences in less than six hours, and thus, **the answer to the first question is that our mining algorithm is sufficiently scalable to handle big amounts of data with the appropriate threshold values**.

The frequent patterns mined by our algorithm helped us identify ten kinds of program transformations. Table 4.9 shows the identified kinds of program transformations grouped according to their scope. The last two columns of the table show the number of pattern occurrences that we investigated and the number of fruitful pattern occurrences, i.e., those that led to the discovery of the corresponding program transformations. Overall, 32% of pattern occurrences were fruitful. Hence, **our answer to the second question is that our algorithm is effective — it mines patterns that often lead to discovery of high-level program transformations**.

Scope	Identified program transformation	Investigated	Fruitful
Statement	Convert Element to Collection	5	2
Loop	Add a Loop Collector	3	1
	Wrap Loop with Timer	2	1
Method	Add Null Check for a Parameter	5	1
Class	Add a New Enum Element	2	1
	Change and Propagate Field Type	3	1
	Change Field to ThreadLocal	2	1
	Copy Field Initializer	2	1
	Create and Initialize a New Field	4	1
	Move Interface Implementation to Inner Class	6	1

Table 4.9: Identified kinds of program transformations. Column **Investigated** shows the number of the investigated pattern occurrences. Column **Fruitful** shows the number of pattern occurrences that were fruitful.

Figure 4.5: An example of Convert Element to Collection transformation.

Note that although we did not have any knowledge of the codebase, we were able to identify and name ten kinds of high-level program transformations. We believe that the original developers of the code could have recognized more transformations. Thus, the result for effectiveness of our mining algorithm is the lower bound.

In the following, we present the discovered transformation kinds in more detail.

Convert Element to Collection. This is a statement-level transformation in which a developer converts a field, parameter, or a local variable of a certain type into a collection (e.g., list, set, array, etc.) of that type. Figure 4.5 shows an example of Convert Element to Collection transformation. Note that if variable `myLock` in Figure 4.5 has any references in the code, then converting it to a collection would require updating all its references accordingly.

Add a Loop Collector. This is a transformation in which a developer introduces a new variable that collects or aggregates the data processed in a loop. Figure 4.6 shows an example of Add a Loop Collector transformation.

Wrap Loop with Timer. A developer applies this transformation to compute the execution time of a loop. The developer surrounds the loop with variables that hold the time before and after the loop execution and outputs the time difference. Figure 4.7 shows an example of Wrap Loop with Timer transformation.

Add Null Check for a Parameter. This is a transformation in which a developer adds `null` precondition checks to all methods of a class that receive a particular parameter such that the methods' bodies

Before

```
for (A a : collection) {
  ...
}
```

After

```
Set<A> collector = new HashSet<A>();
for (A a : collection) {
  ...
  collector.add(a);
}
```

Figure 4.6: An example of Add a Loop Collector transformation.

Before

```
for (...) {
  ...
}
```

After

```
long start = System.currentTimeMillis();
for (...) {
  ...
}
long end = System.currentTimeMillis();
output(end - start);
```

Figure 4.7: An example of Wrap Loop with Timer transformation.

are not executed if the parameter is `null`. Figure 4.8 shows an example of Add Null Check for a Parameter transformation.

Add a New Enum Element. Adding a new element to `enum` triggers a ripple of changes such as adding new `switch` cases, `if-then-else` chains, and dealing with any duplicated code that uses the updated `enum`. Figure 4.9 shows an example of Add a New Enum Element transformation.

Change and Propagate Field Type. This is a transformation in which a developer changes the type of a field. As a result, the developer also has to update the type of some local variables as well as the return type of some methods. Figure 4.10 shows an example of Change and Propagate Field Type transformation.

Change Field to ThreadLocal. To improve thread safety of an application, a developer may decide to convert some fields to `ThreadLocal`. Besides changing the type and the initialization of the converted field, the developer also has to modify all field's accesses such that they use `get()` and `set()` of `ThreadLocal`. Figure 4.11 shows an example of Change Field to ThreadLocal transformation.

Before

```
void m1(P1 p1, P2 p2) {
  ...
}
void m2(P1 p1, P3 p3) {
  ...
}
```

After

```
void m1(P1 p1, P2 p2) {
  if (p1 == null) return;
  ...
}
void m2(P1 p1, P3 p3) {
  if (p1 == null) return;
  ...
}
```

Figure 4.8: An example of Add Null Check for a Parameter transformation.

Before

```
enum E {e1, ... , en};
switch (e) {
  case 1: ...
  ...
  case n: ...
}

if (m.isE1()) { ... }
...
if (m.isEn()) { ... }

P createP() {
  P p = new P();
  p.add(E.e1);
  ...
  p.add(E.en);
}
```

After

```
enum E {e1, ... , en, en+1};
switch (e) {
  case 1: ...
  ...
  case n: ...
  case n+1: ...
}

if (m.isE1()) { ... }
...
if (m.isEn()) { ... }
if (m.isEn+1()) { ... }

P createP() {
  P p = new P();
  p.add(E.e1);
  ...
  p.add(E.en);
  p.add(E.en+1);
}
```

Figure 4.9: An example of Add a New Enum Element transformation.

Before

```
A field;
public A method() {
   ...
  A localVar;
   ...
}
```

After

```
B field;
public B method() {
   ...
  B localVar;
   ...
}
```

Figure 4.10: An example of Change and Propagate Field Type transformation.

Before

```
class A {
  B f;
  void m() {
    int v = compute(f);
    ...
  }
}
```

After

```
class A {
  ThreadLocal<B> f;
  void m() {
    int v = compute(f.get());
    ...
  }
}
```

Figure 4.11: An example of Change Field to ThreadLocal transformation.

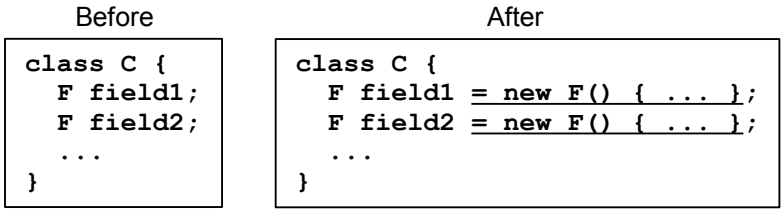


Figure 4.12: An example of Copy Field Initializer transformation.

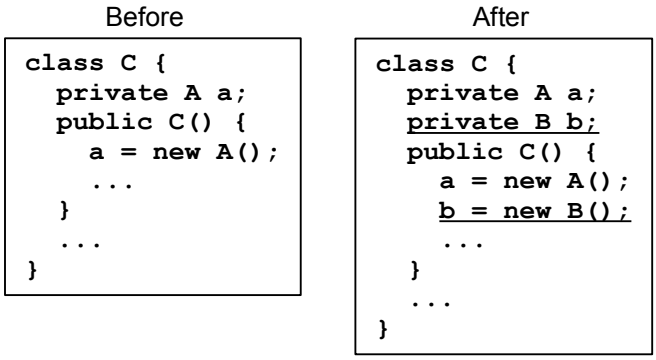


Figure 4.13: An example of Create and Initialize a New Field transformation.

Copy Field Initializer. This is a transformation in which a developer copies the same initializer to several fields. Figure 4.12 shows an example of Copy Field Initializer transformation.

Create and Initialize a New Field. When a developer adds a new field, it has to be properly initialized alongside the already present fields in constructors and other initialization places (e.g., static initialization blocks). Figure 4.13 shows an example of Create and Initialize a New Field transformation.

Move Interface Implementation to Inner Class. This is a transformation that describes a scenario in which a developer moves the implementation of an interface from a class to its newly created inner class. Figure 4.14 shows an example of Move Interface Implementation to Inner Class transformation.

The mined code change patterns helped us identify ten kinds of interesting program transformations whose scopes range from individual statements to whole classes. Thus, **our answer to the third question**

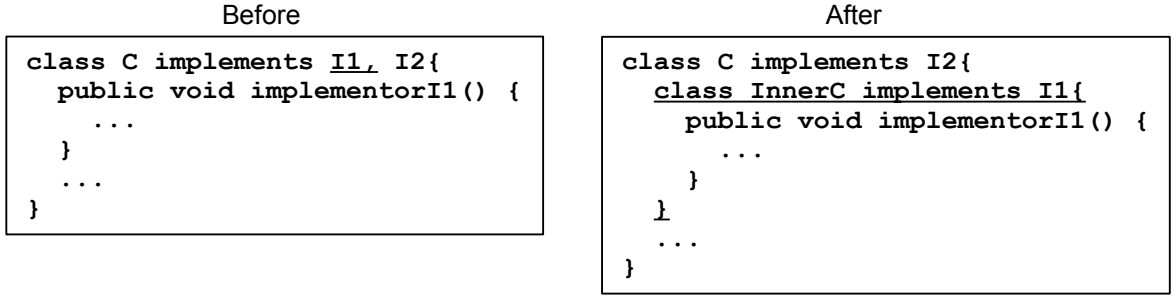


Figure 4.14: An example of Move Interface Implementation to Inner Class transformation.

is that our algorithm is useful.

4.5.3 Threats to Validity

In our experiment, we used the output of the AST node operations inference algorithm (see Section 2.2) to prepare the input to our frequent code change pattern mining algorithm. Consequently, imprecisions in the inferred AST node operations could negatively affect our mining results. Note, however, that our approach to mining frequent code change patterns is independent of the way its input is produced.

We investigated the mined patterns manually, and thus, might have missed some of their corresponding high-level program transformations. Also, we investigated only a fraction of the mining results. However, our experiment did not aim at discovering all program transformations performed by our participants. Instead, our goal was to show that our algorithm mines patterns that effectively point to high-level program transformations, and we believe that discovering several such transformations in a reasonable amount of time (identifying and documenting these transformations took just a couple of days) supports this claim.

In our study, we collected code evolution data from developers who use Eclipse for Java programming. Consequently, the identified high-level program transformations might not be generalizable to other programming environments and languages. Nevertheless, our approach to identifying such transformations is orthogonal to the way developers make their code changes.

Our dataset is not publicly available due the nondisclosure agreement with our participants.

Chapter 5

Related Work

5.1 Program Transformations

A program transformation is any change that converts one program into another one. Developers transform programs daily, improving their design or performance, fixing bugs, implementing new features, etc. A variety of automated program transformation tools were developed to support these programming activities. For example, text substitution is one of the simplest automated program transformations. It is supported by all modern Integrated Development Environments (IDEs) as well as by some specialized command-line tools (e.g., `sed` and `awk` [36]).

Behavior-preserving program transformations keep the observable behavior of the transformed program intact. Although behavior-preserving source-to-source transformations are the most common example of compiling a program, the executable source code may be produced by transforming the original program that is expressed in a variety of ways, e.g., as high-level design patterns [18], as algebraic specifications [69,72], etc.

Performance tuning transformations are behavior-preserving transformations that improve performance of the underlying code. Such transformations are implemented as part of compilers starting with IBM's Fortran I compiler in 1959 [3], which supported constant folding, common subexpression elimination, loop-independent code motion, etc. With the advancements in user interfaces, program transformation tools became interactive. A developer directs a tool to apply a particular transformation to a specific part of the code, while the tool automatically transforms the corresponding code and (usually) checks the correctness of the transformation. Many interactive source-to-source transformation tools were created to help developers parallelize their programs, e.g., PTOOL [4], Faust [56], ParaScope [8, 25, 58, 75, 76], and D [65].

Refactorings are a different kind of source-to-source behavior-preserving transformations. Refactorings help developers improve the design of the underlying program rather than its performance. Opdyke and Johnson were the first to mention the term "refactoring" in the literature in 1990 [111]. Bill Griswold described several refactorings (e.g., rename, extract, inline, etc.) and implemented them in a prototype

tool for Scheme as part of his PhD thesis in 1991 [54]. Bill Opdyke identified refactorings for developing object-oriented frameworks and implemented them in a prototype tool for C++ as part of his PhD thesis in 1992 [110]. Brant and Roberts developed the first production-quality refactoring tool, Smalltalk Refactoring Browser [122], which was also the first refactoring tool fully integrated in a development environment, Smalltalk. The increasing popularity of eXtreme Programming (XP) [13], which relies on refactoring as an important step during code development, as well as the release of the Martin Fowler's book on refactoring in 1999 [45] led to wide acceptance of refactorings among software developers.

As part of our prior work, we developed a range of program analysis and transformation tools: ReBA [35] adapts client applications to new versions of evolving libraries; AMPIzer [105, 112, 147] transforms legacy Message Passing Interface (MPI) applications to make them executable on Adaptive MPI (AMPI), whose features benefit the transformed applications in many ways (e.g., communication/computation overlap, dynamic load balancing, etc.); SOTER [103] detects message contents that can be safely passed by reference in Actor Model programs in order to improve their performance. Also, we developed CodingSpectator [132,133] and employed it in a field study to better understand how developers interact with automated refactoring tools provided by Eclipse.

5.2 Representation of Program Transformations

One of the simplest approaches to program transformation is to treat the source code of a program as plain text and transform it by applying text substitution rules, which can be represented as regular-expression patterns [14]. This simple representation does not consider the structure of the underlying program, and thus, it is not capable to express more sophisticated transformations. Besides, if a transformation is big, such representation becomes cumbersome to understand and maintain.

More advanced representations take into account the structural properties of the underlying code. For example, A* [85] and TAWK [55] allow direct manipulations on the Abstract Syntax Tree (AST) of the transformed program. Mens et al. [95] express refactorings as graph operations. Stratego/XT [136] employs even higher level of abstraction and represents transformations as term rewrite rules. These tools are powerful and expressive, but to be applied properly, they require an extensive expertise on the part of a programmer.

To simplify representation of program transformations, while preserving its expressiveness, some tools extend the syntax of the underlying language, e.g., DMS [12], REFINE [19], TXL [26]. Although this approach improves readability and maintainability of transformations, it requires a significant effort from a programmer and consequently, it is rarely applied in practice. Several other tools augment the underlying

language with different annotations [7, 23, 74, 123]. This approach suffers from an additional burden to write and maintain transformations as part of the program's source code.

5.3 Capturing Fine-grained Code Changes

Robbes et al. [119, 121] proposed to make changes first-class citizens and capture them directly from an IDE as soon as they happen. They developed a tool, SpyWare [120], that implements these ideas. SpyWare gets notified by the Smalltalk compiler in Squeak IDE whenever the AST of the underlying program changes. SpyWare records the captured AST modification events as operations on the corresponding AST nodes. Also, SpyWare records automated refactoring invocation events.

Although our work is inspired by similar ideas, our tool, CodingTracker, significantly differs from SpyWare. CodingTracker captures raw fine-grained code edits rather than a compiler's AST modification events. The recorded data is so precise that CodingTracker is able to replay it in order to reproduce the exact state of the evolving code at any point in time. Also, CodingTracker implements a novel AST node operations inferencing algorithm that does not expect the underlying code to be compilable or even fully parsable. Besides, CodingTracker captures a variety of evolution data that does not represent changes to the code, e.g., interactions with VCS, application and test runs, etc.

Sharon et al. [38] implemented EclipsEye, porting some ideas behind SpyWare to Eclipse IDE. Similarly to SpyWare, EclipsEye gets notified by Eclipse about AST changes in the edited application, but these notifications are limited to the high-level AST nodes starting from fields/methods and up.

Omori and Maruyama [108, 109] developed a similar fine-grained operation recorder and replayer for the Eclipse IDE. In contrast to CodingTracker, their tool does not infer AST node operations but rather associates code edits with AST nodes, to which they might belong. Besides, CodingTracker captures more operations such as those that do not affect code like runs of programs and tests and version control system operations. The additional events that CodingTracker captures enabled us to study the test evolution patterns and the degree of loss of code evolution information in version control systems.

Yoon et al. [142] developed a tool, Fluorite, that records low-level events in Eclipse IDE. Fluorite captures sufficiently precise fine-grained data to reproduce the snapshots of the edited files. But the purpose of Fluorite is to study code editing patterns rather than program transformations in general. Therefore, Fluorite does not infer AST node operations from the collected raw data. Also, it does not capture such important evolution data as interactions with VCS, test runs, or effects of automated refactorings.

Chan et al. [22] proposed to conduct empirical studies on code evolution employing fine-grained revision

history. They produce fine-grained revision history of an application by capturing the snapshots of its files at every save and compilation action. Although such a history contains more detailed information about an application's code evolution than a common VCS, it still suffers from the limitations specific to snapshot-based approaches, in particular, the irregular intervals between the snapshots and the need to reconstruct the low-level changes from the pairs of consecutive snapshots.

5.4 Inferring High-Level Program Transformations

5.4.1 Detecting Structural Changes

To provide greater insight into source code evolution, researchers have proposed tools to reconstruct high-level source code changes (e.g., operations on AST nodes, refactorings, restructurings, etc.) from the coarse-grained data supplied through VCS snapshots.

Fluri et al. [43] proposed a change distilling algorithm to extract fine-grained changes from two snapshots of a source code file and implemented this algorithm in a tool, ChangeDistiller. ChangeDistiller represents the difference between two versions of a file as a sequence of atomic operations on the corresponding AST nodes. We also express changes as AST node operations, but our novel algorithm infers them directly from the fine-grained changes produced by a developer rather than from snapshots stored in VCS.

Kim et al. [79] proposed summarizing the structural changes between different versions of a source code file as high-level *change rules*. Change rules provide a cohesive description of related changes beyond deletion, addition and removal of a textual element. Based on this idea, they created a tool that automatically infers those change rules and presents them as concise and understandable transformations to the programmer.

All these as well as many other [5, 52, 78, 82, 134, 139] tools designed to detect structural changes in the evolving code operate on VCS snapshots. However, the results of our field study presented in Section 2.3 show that VCS snapshots provide incomplete and imprecise data, thus compromising the accuracy of these tools. In a change-oriented programming environment, we overcome the limitations of file-based VCSs by inferring high-level program transformations from fine-grained code changes recorded by CodingTracker.

5.4.2 Automated Inference of Refactorings

In an early work, Demeyer et al. [31] inferred refactorings by comparing two different versions of source code using heuristics based only on low-level software metrics — method size, class size, and inheritance levels. Kim et al. [82] used a function similarity algorithm to detect methods that have been renamed. More recent refactoring inference approaches detect refactorings depending on how well they match a set of *characteristic*

properties that are constructed from the differences between two consecutive versions of an application. Dig et al. [32] employed references of program entities like instantiations, method calls, and type imports as their set of characteristic properties. Weißgerber and Diehl [137] used characteristic properties based on names, signature analysis, and clone detection. More recently, Prete et al. [115] devised a template-based approach that can infer up to 63 of the 72 refactorings cataloged by Fowler [45]. Their templates build upon characteristic properties such as accesses, calls, inherited fields, etc., that model code elements in Java. Their tool, Ref-Finder, infers the widest variety of refactorings to date.

All these approaches rely exclusively on snapshots from VCS to infer refactorings. Thus, the accuracy of detection depends on the closeness of the two snapshots being compared. We have shown (see Section 3.3.3, **RQ7**) that many refactorings are shadowed and do not ever reach a commit. This compromises the accuracy of inference algorithms that rely on snapshots. Moreover, snapshot-based approaches (with the exception of Ref-Finder) usually concentrate only on API-level changes leaving out many of the completely or partially local refactorings that we infer (see Table 3.1). This paints an incomplete picture of the evolution of the code.

To address such inadequacies, our inference algorithm leverages fine-grained edits. Similar to existing approaches, our algorithm infers refactorings by *matching* a set of characteristic properties for each refactoring (see Table 3.5). Our properties consist of high-level semantic changes such as adding a field, deleting a variable, etc. In contrast to existing approaches, our properties are precise because they are constructed directly from the AST operations that are recorded on each code edit.

In parallel with our tool, Ge et al. [49] developed BeneFactor and Foster et al. [44] developed WitchDoctor. Both these tools continuously monitor code changes to detect and complete manual refactorings in *real-time*. Although conceptually similar, our tools have different goals — we infer complete refactorings, while BeneFactor and WitchDoctor focus on inferring and completing partial refactorings in real time. Thus, their tools can afford to infer fewer kinds of refactorings and with much lower accuracy. Nonetheless, while orthogonal to our work on studying code evolution, these projects highlight the potential of using refactoring inference algorithms based on fine-grained code changes to improve the IDE. In the following, we compare our tool with the most similar tool, WitchDoctor, in more detail.

Like our tool, WitchDoctor represents fine-grained code changes as AST node operations and uses these operations to infer refactorings. Although similar, the AST node operations and refactoring inference algorithms employed by WitchDoctor and our tool have a number of differences. In particular, our AST node operations inference algorithm (see Section 2.2) employs a range of heuristics for better precision, e.g., it handles Eclipse's linked edits and jumps over the unparsable state of the underlying code. Witch-

Doctor specifies refactorings as requirements and constraints. Our refactoring inference algorithm defines refactorings as collections of properties without explicitly specifying any constraints on them. Instead, the properties' attributes matching ensures compatibility of the properties that are part of the same refactoring (see Section 3.2.2). Additionally, our algorithm infers migrated AST nodes and refactoring fragments, which represent a higher level of abstraction than properties that are constructed directly from AST node operations. The authors of WitchDoctor focused on real-time performance of their tool. Since we applied our tool off-line, we were not concerned with its real-time performance, but rather assessed both precision and recall of our tool on real world data.

5.5 Mining Frequent Itemsets

The major challenge in mining frequent itemsets is to develop scalable algorithms that can effectively handle large transaction databases. One of the fundamental distinctions between different approaches to mining frequent itemsets is whether mining is performed with or without candidate generation. Agrawal et al. [2] observed that an n-itemset is frequent only if all its subsets are also frequent. Their mining algorithm, Apriori, leverages this property by using frequent n-itemsets to generate $(n+1)$-itemset candidates. Apriori checks the newly generated candidates against the transaction database to establish those of them that are frequent. The algorithm starts with detecting frequent 1-itemsets directly from the transaction database and proceeds iteratively until no more frequent itemsets can be found.

Mining with candidate generation has two major drawbacks: a) it generates redundant itemsets that are found to be infrequent; b) it repeatedly scans the transaction database while progressing through the iterations. Mining without candidate generation addresses both these limitations. Such mining can be broadly divided into mining using *horizontal data format* and mining using *vertical data format*. The horizontal data format represents a transaction database as a set of tuples $< TransactionID, itemset >$. Table 4.2 illustrates a database in this format. Han et al. [60] suggested to mine frequent itemsets without candidate generation using horizontal data format. Their frequent-pattern growth (FP-growth) algorithm first scans the database to detect frequent 1-itemsets. The algorithm uses these 1-itemsets to construct FP-tree, an extended prefix-tree structure. Then, the algorithm expands the initial FP-tree by growing pattern fragments in a recursive fashion.

Zaki [144] proposed a different approach to mining frequent itemsets without candidate generation. His algorithm, Eclat, explores the vertical data format, which explicitly stores transactions' identifiers (tidsets) for every itemset. Eclat computes $(n+1)$-itemsets from n-itemsets by intersecting their tidsets. The

algorithm collects the initial set of frequent 1-itemset by scanning the transaction database.

Subsequently, Zaki [145] developed CHARM, a more advanced algorithm for mining data in the vertical data format. The algorithm is based on the same idea of intersecting itemsets' tidsets to produce new itemsets, but it specifies the search problem using the notion of itemset-tidset tree (IT-tree). Also, CHARM introduces several optimizations, including the search for closed itemsets.

All the approaches above operate on a database with disjoint transactions, each containing a set of items. On the contrary, our frequent code change pattern mining algorithm handles *overlapping* transactions and *itembags* rather than itemsets, which are the two major challenges specific to frequent code change pattern mining from *continuous* sequence of code changes.

5.6 Mining Source Code

Source code mining research has a long history. Here, we present several representative examples.

Michail [96] applied data mining techniques to detect how a library is reused in different applications. The mined library reuse patterns, represented as association rules, facilitate the reuse of the library components by developers.

Li et al. [90] employed frequent itemset mining to extract programming rules from the source code of an application. They also showed that source code fragments that violate the extracted rules are likely to be buggy.

Holmes et al. [66] matched the structural context of the edited source code against a code repository to present a developer with the examples demonstrating the relevant API usage. Similarly, Bruch et al. [17] proposed to improve the IDE's code completion systems by making them learn from code repositories.

Hovemeyer et al. [67] developed FindBugs, a tool that detects a variety of bug patterns in an application by statically matching bug pattern descriptions against the underlying source code. More recently, Lin et al. [92] proposed an approach to search for a specific kind of bug patterns — violations of check-then-act idioms.

All these approaches mine the application's source code, while our algorithm mines code changes, and thus, it can identify new patterns *on-the-fly*.

Another direction of research is mining source code change patterns from the Version Control System (VCS) history of an application. Ying et al. [141] and Zimmermann et al. [150] apply data mining techniques on the application's revision history to detect software artifacts (e.g., methods, classes, etc.) that are usually changed together. The mined association rules predict what other source code locations a developer needs

to consider while performing a particular change. Uddin et al. [130] proposed to mine VCS histories of client applications to study how their use of APIs evolves over time, which is helpful both to developers and users of the libraries' APIs. Canfora et al. [20, 21] and Thummalapenta et al. [128] used VCS snapshots to study and track the evolution of different software entities such as source lines, bugs, and clones. In the domain of software testing, Zaidman et al. [143] mined software repositories to explore how production and test code co-evolve.

Mining VCS snapshots of an application is exposed to the limited nature of VCS data. In our empirical study of code evolution (see Section 2.3), we showed that data stored in VCS is *imprecise*, *incomplete*, and makes it *impossible* to perform analysis that involves the time dimension inside a single VCS snapshot. Also, similarly to other source code mining techniques, these approaches are not suitable for identifying code change patterns on-the-fly.

Wit et al. [30] performed live monitoring of the clipboard to detect clones. Their tool tracked the detected clones, offering several resolution strategies whenever the clones were edited inconsistently. Our approach of detecting similar changes to different parts of the code is complementary to detecting different changes to similar parts of the code.

5.7 Empirical Studies on Source Code Evolution

Early work on source code evolution relied on the information stored in VCS as the primary source of data. The lack of fine-grained data constrained researchers to concentrate mostly on extracting high-level metrics of software evolution, e.g., number of lines changed, number of classes, etc.

Eick et al. [39] identified specific indicators for *code decay* by conducting a study on a large (∼100,000,000 LOC) real time software system for telephone systems. These indicators were based on a combination of metrics such as number of lines changed, commit size, number of files affected by a commit, duration of a change and the number of developers contributing to a file.

Xing et al. [138] analyzed the evolution of design in object-oriented software by reconstructing differences between snapshots of software releases at the UML level using their tool, UMLDiff. UML level changes capture information at the class level and can be used to study how classes, fields, and methods have changed from each version. From these differences, they tried to identify distinct patterns in the software evolution cycles.

Gall et al. [46] studied the logical dependencies and change patterns in a product family of Telecommunication Switching Systems by analyzing 20 punctuated software releases over two years. They decomposed

the system into modules and used their CAESAR technique to analyze how the structure and software metrics of these modules evolved through different releases.

For these kinds of analyses, the data contained in traditional VCS is adequate. However, for more interesting analyses that require program comprehension, relying only on high-level information from VCS is insufficient. In particular, Robbes in his PhD thesis [118, p.70] shows the difference in the precision of code evolution analysis tools applied to fine-grained data vs. coarse-grained VCS snapshots. This client level comparison is complementary to our work, in which we quantify the extent of data loss and imprecision in VCS snapshots independently of a particular client tool.

5.8 Empirical Studies on the Practice of Refactoring

Xing and Stroulia [140] report that 70% of all changes observed in the evolution of the Eclipse code base are expressible as refactorings. Dig and Johnson [33] studied four open source frameworks and one library and concluded that more than 80% of component API evolution is expressible through refactorings. These studies indicate that the practice of refactoring plays a vital role in software evolution and is an important area of research.

Work on empirical research on the usage of automated refactoring tools was stimulated by Murphy et al.'s [98] study of 41 developers using the Java tools in Eclipse. Their study provided the first empirical ranking of the relative popularities of different automated refactorings, demonstrating that some tools are used more frequently than others. Subsequently, Murphy-Hill et al.'s [100] study on the use of automated refactoring tools provided valuable insights into the use of automated refactorings in the wild by analyzing data from multiple sources.

Due to the non-intrusive nature of CodingTracker, we were able to deploy our tool to more developers for longer periods of time, providing a more complete picture of refactoring in the wild. We inferred and recorded an *order* of magnitude more *manual* refactorings than Murphy-Hill et al. in their sampling-based approach. Murphy-Hill sampled 80 commits from 12 developers for a total of 261 refactoring invocations whereas our tool recorded 1,520 hours from 23 developers for a total of 5,371 refactoring invocations.

Murphy-Hill et al.'s [100] study found that (i) refactoring tools are underused and (ii) the kinds of refactorings performed manually are different from those performed using tools. Our data (see Section 3.3.3, **RQ3**) corroborates both these claims. We found that some refactorings are performed manually more frequently, even when the automated tool exists and the developer is aware of it. Due to the large differences in the data sets (261 from Murphy-Hill et al. vs. 5,371 in our study), it is infeasible to meaningfully compare

the raw numbers for each refactoring kind, while comparing the ratios would not yield statistically significant results. Our work also builds upon their work by providing a more detailed breakdown of the manual and automated usage of each refactoring tool according to different participants' behavior.

Vakilian et al. [131] observed that many advanced users tend to compose several refactorings to achieve different purposes. Our results about clustered refactorings (see Section 3.3.3, **RQ6**) provide additional empirical evidence of such practices.

Chapter 6

Future Work

6.1 Accessing and Manipulating Code Changes

Currently, CodingTracker's replayer enables a developer to browse fine-grained code changes applied to an application. Although a developer can apply filters to focus on a particular kind (or kinds) of code changes, the list of changes is still too big for effective comprehension. To improve readability of code changes, we plan to (i) implement grouping mechanisms that would represent changes in a hierarchical way; (ii) add a feature that would map code fragments to the corresponding changes and vice-versa, thus allowing a developer to focus on changes that pertain to a code fragment of interest (e.g., a particular method or loop); and (iii) add graphical views that would visually represent high-level information about code changes, e.g., showing intensity of changes over time.

Additionally, we would like to enable a developer to manipulate the code changes directly. In particular, we plan to allow a developer to undo or tweak a specific change in the list, e.g., adjust a configuration option of a refactoring. Also, we plan to make changes re-applicable, i.e., a developer would be able to choose a particular change (or changes) from the list and apply them to a different code fragment.

6.2 On-Line Refactoring Assistance

Although our refactoring inference algorithm operates on continuous code changes, and thus, can be applied on-line, in our empirical study, we applied the algorithm while replaying the previously captured code changes. Nevertheless, we plan to make our inference algorithm part of Eclipse IDE such that it can assist developers with the refactoring process on-the-fly. For example, after inferring a manual refactoring performed by a developer, our tool will check its correctness and report to the developer any detected inconsistencies. Even when there are no problems to report, our tool will notify the developer that the corresponding refactoring could have been applied automatically, thus increasing the developer's awareness of the automated refactorings supported by Eclipse.

Similarly to BeneFactor [49] and WitchDoctor [44], we will extend our tool to infer partial refactorings. Consequently, our tool will be able to assist a developer while a manual refactoring is in progress. For example, our tool could offer the developer to complete the refactoring automatically.

6.3 Improved Mining of Frequent Code Change Patterns

Although our current frequent code change pattern mining algorithm helps identify high-level program transformations, we would like to minimize the manual part of this identification. We plan to implement postprocessors of the mining results that would simplify identification of program transformations. In particular, we are going to employ grouping of the results that would organize the mined patterns according to their equivalence classes with respect to the corresponding program transformations. Also, we plan to introduce specialized filters that would prioritize patterns containing particular combinations of AST node operations. In the long term, we would like to incorporate our mining algorithm into an IDE and completely eliminate the manual part of high-level program transformations identification.

Another line of work enabled by our tool is to (i) implement specific inference algorithms for the program transformations that we identified using our mining algorithm, and (ii) automate the identified transformations in an IDE. We believe that specialized inference algorithms are more effective than any agnostic inference algorithm could be. Consequently, such algorithms would allow researchers to assess with a high degree of confidence the popularity of different high-level transformations. Then, tool builders could pick the most popular transformations and automate them in an IDE.

6.4 Extended Empirical Studies

Our empirical studies on source code evolution and on the practice of refactorings serve as mere examples of the kinds of research questions that a researcher can answer using the detailed and precise code evolution data collected and processed by CodingTracker. We think that the already collected data can be used for further studies on how developers change their code. In particular, we plan to analyze the correlations between such programming activities as refactoring and testing, refactoring and interactions with VCS, etc.

CodingTracker captures code evolution data in a very unobtrusive way. Therefore, we think that it could be made part of the official Eclipse distribution and collect huge amounts of data from many developers (similarly to the current Eclipse data collectors, a developer would have an option to not upload the collected data). Collecting more data would not only help to answer research questions more precisely, but also would enable us to answer new kinds of research questions. For example, we would be able to compare code evolution

activities for different projects and settings (e.g., open source vs. proprietary, industry vs. academia, etc.). Also, collecting code evolution data for longer time periods would help us study different phases of software development of a given project (e.g., before vs. after a release) as well as identify long term trends in the project's evolution.

Chapter 7

Conclusions

Although change is the heart of software development, the traditional programming environments and toolkits treat changes as a by-product of software evolution. In this dissertation, we show that placing change at the center of a programming environment benefits code evolution researchers, tool builders, and application developers.

We developed CodingTracker, an infrastructure that captures and replays detailed and precise changes to the code, represents the captured raw code edits uniformly as Abstract Syntax Tree (AST) node operations, infers refactorings from *continuous* code changes, and mines frequent code change patterns, thus helping researchers identify *previously unknown* high-level program transformations. We employed CodingTracker to conduct an empirical study on source code evolution, in which we quantified the limitations of the data stored in Version Control Systems (VCS). CodingTracker also enabled us to conduct an extensive empirical study on the practice of manual vs. automated refactorings.

The primary source of data in code evolution research is the file-based VCS. The results of our empirical study show that although popular among researchers, a file-based VCS provides data that is incomplete and imprecise. Moreover, many interesting research questions that involve code changes and other development activities (e.g., automated refactorings or test runs) require evolution data that is not captured by VCS at all. We found that 37% of code changes are *shadowed* by other changes, and are not stored in VCS. Thus, VCS-based code evolution research is incomplete. Second, programmers intersperse different kinds of changes in the same commit. For example, 46% of refactored program entities are also edited in the same commit. This overlap makes the VCS-based research imprecise. The data collected by CodingTracker enabled us to answer research questions that could not be answered using VCS data alone. In particular, we discovered that 40% of test fixes involve changes to the tests, 24% of changes committed to VCS are untested, and 85% of changes to a method during an hour interval are clustered within 15 minutes. These results confirm that more detailed data than what is stored in VCS is needed to study software evolution accurately.

There are many ways to learn about the practice of refactoring, such as observing and reflecting on one's

own practice, observing and interviewing other practitioners, and controlled experiments. But an important way is to analyze the changes made to a program, since programmers' beliefs about what they do can be contradicted by the evidence. Thus, it is important to be able to analyze programs and determine the kind of changes that have been made. This is traditionally done by looking at the difference between snapshots. Our empirical study on source code evolution shows that VCS snapshots lose information. Moreover, VCS snapshots do not allow researchers to distinguish manual and automated refactorings. Therefore, to study the practice of refactoring, we employed a continuous analysis of change. We discovered that refactorings tend to be clustered, that programmers often change the name of a program entity several times within a short period of time and perform more manual than automated refactorings. Continuous analysis is better at detecting refactorings than analysis of snapshots, and it ought to become the standard for detecting refactorings.

Although mining frequent code change patterns has a long research history, we are the first to present an algorithm that mines such patterns from a *continuous* sequence of code changes rather than from the static source code. Our algorithm effectively handles *overlapping* transactions that contain multiple instances of the same item kind — the major challenge that distinguishes our approach from the existing frequent itemset mining techniques. Since our algorithm mines code changes continuously, it can be applied either *on-the-fly* or on a previously recorded sequence of code changes. To evaluate our algorithm, we used 1,520 hours of real world code changes that we collected from 23 developers. We showed that our mining algorithm is scalable, effective, and useful. Analyzing some of the mining results, we identified ten popular kinds of high-level program transformations.

References

[1] B. Adams, Z. M. Jiang, and A. E. Hassan. Identifying crosscutting concerns using historical code changes. In *ICSE*, 2010.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[3] F. E. Allen. A technological review of the FORTRAN I compiler. In *Proceedings of the June 7-10, 1982, national computer conference*. ACM, 1982.

[4] R. Allen, D. Bumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *ICPP*, 1986.

[5] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE*, 2004.

[6] Apache Gump continuous integration tool. http://gump.apache.org/.

[7] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. OOPSLA, 2005.

[8] V. Balasundaram, K. Kennedy, U. Kremer, K. McKinley, and J. Subhlok. The parascope editor: an interactive parallel programming tool. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, 1989.

[9] Bamboo Continuous Integration and Release Management. http://www.atlassian.com/software/bamboo/.

[10] R. D. Banker, S. M. Datar, C. F. Kemerer, and D. Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36, 1993.

[11] J. Bansiya. *Object-Oriented Application Frameworks: Problems and Perspectives*, chapter Evaluating application framework architecture structural and functional stability. 1999.

[12] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. ICSE, 2004.

[13] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[14] A. F. Blackwell. *SWYN: a visual representation for regular expressions*. Morgan Kaufmann Publishers Inc., 2001.

[15] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr. Code Bubbles: rethinking the user interface paradigm of integrated development environments. In *ICSE*, 2010.

[16] F. P. Brooks, Jr. *The mythical man-month (anniversary ed.)*. 1995.

[17] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *FSE*, 2009.

[18] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Syst. J.*, 1996.

[19] S. Burson, G. Kotik, and L. Markosian. A program transformation approach to automating software re-engineering. COMPSAC, 1990.

[20] G. Canfora, M. Ceccarelli, L. Cerulo, and M. D. Penta. How long does a bug survive? an empirical study. In *WCRE*, 2011.

[21] G. Canfora, L. Cerulo, and M. D. Penta. Tracking your changes: A language-independent approach. In *IEEE Software*, 2009.

[22] J. Chan, A. Chu, and E. Baniassad. Supporting empirical studies by non-intrusive collection and visualization of fine-grained revision history. In *Proceedings of the 2007 OOPSLA workshop on Eclipse technology eXchange*, 2007.

[23] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. ICSM, 1996.

[24] S. D. Conte, H. E. Dunsmore, and V. Y. Shen. *Software engineering metrics and models*. 1986.

[25] K. D. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The parascope parallel programming environment. In *Proceedings of the IEEE*, 1993.

[26] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 1991.

[27] CVS - Concurrent Versions System. http://cvs.nongnu.org/.

[28] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA*, 2010.

[29] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. ReAssert: Suggesting repairs for broken unit tests. In *ASE*, 2009.

[30] M. de Wit, A. Zaidman, and A. van Deursen. Managing code clones using dynamic change tracking and resolution. In *ICSM*, 2009.

[31] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA*, 2000.

[32] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *ECOOP*, 2006.

[33] D. Dig and R. Johnson. The role of refactorings in API evolution. In *ICSM*, 2005.

[34] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *Journal of Soft. Maint. and Evol.: Research and Practice*, 2006.

[35] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *ICSE*, 2008.

[36] D. Dougherty and A. Robbins. *sed & awk*. O'Reilly & Associates, Inc., 1997.

[37] Eclipse - The Eclipse Foundation open source community website. http://www.eclipse.org/.

[38] EclipsEye - Spying on Eclipse. http://www.inf.usi.ch/faculty/lanza/Downloads/Shar07a.pdf.

[39] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *TSE*, 27:1–12, 2001.

[40] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2, 2000.

[41] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of identifier renamings. In *MSR*, 2011.

[42] M. Feathers. *Working Effectively with Legacy Code*. 2004.

[43] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *TSE*, 33:725–743, November 2007.

[44] S. Foster, W. G. Griswold, and S. Lerner. WitchDoctor: IDE Support for Real-Time Auto-Completion of Refactorings. In *ICSE*, 2012.

[45] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[46] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *ICSM*, 1998.

[47] H. Gall, M. Jazayeri, R. R. Klsch, and G. Trausmuth. Software evolution observations based on product release history. In *ICSM*, 1997.

[48] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *IWMPSE*, 2003.

[49] X. Ge, Q. L. DuBose, and E. Murphy-Hill. Reconciling manual and automatic refactoring. In *ICSE*, 2012.

[50] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *ICSM*, 2004.

[51] Git - the fast version control system. http://git-scm.com/.

[52] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31:166–181, 2005.

[53] C. Gorg and P. Weisgerber. Detecting and visualizing refactorings from software archives. In *ICPC*, 2005.

[54] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[55] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the 4th International Workshop on Program Comprehension*, WPC, 1996.

[56] V. A. Guarna, Jr., D. Gannon, D. Jablonowski, A. D. Maloney, and Y. Gaur. Faust: An integrated environment for parallel programming. *IEEE Software*, 1989.

[57] T. Guimaraes. Managing application program maintenance expenditures. *Commun. ACM*, 26, 1983.

[58] M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny, and G. Roth. Experiences using the parascope editor: an interactive parallel programming tool. In *PPOPP*, 1993.

[59] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Min. Knowl. Discov.*, 15, 2007.

[60] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[61] S. Hassaine, F. Boughanmi, Y.-G. Guéhéneuc, S. Hamel, and G. Antoniol. A seismology-inspired approach to study change propagation. In *ICSM*, 2011.

[62] A. E. Hassan. Predicting faults using the complexity of code changes. In *ICSE*, 2009.

[63] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *ICSE*, 2005.

[64] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: a taxonomical study of large commits. In *MSR*, 2008.

[65] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D editor: a new interactive parallel programming tool. In *Proceedings of the 1994 conference on Supercomputing*, 1994.

[66] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Trans. Softw. Eng.*, 32, 2006.

[67] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA*, 2004.

[68] Hudson extensive continuous integration server. http://hudson-ci.org/.

[69] S. Jefferson and S. N. Kamin. Executable specifications with quantifiers in the FASE system. In *POPL*, 1986.

[70] Jenkins extendable open source continuous integration server. http://jenkins-ci.org/.

[71] H. Kagdi, M. L. Collard, and J. I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Softw. Maint. Evol.*, 19, March 2007.

[72] S. N. Kamin, S. Jefferson, and M. Archer. The role of executable specifications: The FASE system. In *IEEE Symposium on Application and Assessment of Automated Tools for Software Development*, 1983.

[73] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE*, 2011.

[74] R. Keller and U. Hölzle. Binary component adaptation. ECOOP, 1998.

[75] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Analysis and transformation in the parascope editor. In *Proceedings of the 5th international conference on Supercomputing*. ACM, 1991.

[76] K. Kennedy, K. S. McKinley, and C.-W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 1991.

[77] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of API-level refactorings during software evolution. In *ICSE*, 2011.

[78] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *ICSE*, 2009.

[79] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *ICSE*, 2007.

[80] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *FSE*, 2012.

[81] S. Kim, E. J. W. Jr., and Y. Zhang. Classifying software changes: Clean or buggy? *TSE*, 34(2), 2008.

[82] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proceedings of the 12th Working Conference on Reverse Engineering*. IEEE Computer Society, 2005.

[83] S. Kim, K. Pan, and E. J. Whitehead, Jr. Micro pattern evolution. In *MSR*, 2006.

[84] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead. Automatic identification of bug-introducing changes. In *ASE*, 2006.

[85] D. A. Ladd and J. C. Ramming. A*: a language for implementing language processors. *IEEE Transactions on Software Engineering*, 1995.

[86] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Micro interaction metrics for defect prediction. In *ESEC/FSE*, 2011.

[87] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-drop refactoring: intuitive and efficient program transformation. In *ICSE*, 2013.

[88] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, September 1980.

[89] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[90] Z. Li and Y. Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *FSE*, 2005.

[91] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21, 1978.

[92] Y. Lin and D. Dig. Check-then-act misuse of java concurrent collections. In *ICST*, 2013.

[93] M. Mattson and J. Bosch. Frameworks as components: a classification of framework evolution. In *NWPER*, 1998.

[94] M. Mattson and J. Bosch. Three Evaluation Methods for Object-oriented Frameworks Evolution - Application, Assessment and Comparison. Technical report, University of Karlskrona/Ronneby, Sweden, 1999.

[95] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. Formalizing refactorings with graph transformations. *J. Softw. Maint. Evol.*, 2005.

[96] A. Michail. Data mining library reuse patterns in user-selected applications. In *ASE*, 1999.

[97] M. Mirzaaghaei, F. Pastore, and M. Pezze. Automatically repairing test cases for evolving method declarations. In *ICSM*, 2010.

[98] G. C. Murphy, M. Kersten, and L. Findlater. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software*, 2006.

[99] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: observations and tools for extract method. In *ICSE*, 2008.

[100] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *IEEE TSE*, 2012.

[101] S. Negara, N. Chen, M. Vakilian, R. E. Johnson, and D. Dig. A comparative study of manual and automated refactorings. In *ECOOP*, 2013.

[102] S. Negara, M. Codoban, D. Dig, and R. E. Johnson. Mining continuous code changes to detect frequent program transformations. Technical Report http://hdl.handle.net/2142/43889, University of Illinois at Urbana-Champaign, 2013.

[103] S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *PPoPP*, 2011.

[104] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP*, 2012.

[105] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kalé, and P. M. Ricker. Automatic MPI to AMPI program transformation using Photran. In *Proceedings of the 3rd Workshop on Productivity and Performance*, PROPER, 2010.

[106] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *ICSE*, 2012.

[107] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers. Active code completion. In *ICSE*, 2012.

[108] T. Omori and K. Maruyama. A change-aware development environment by recording editing operations of source code. In *MSR*, 2008.

[109] T. Omori and K. Maruyama. An editing-operation replayer with highlights supporting investigation of program modifications. In *IWMPSE-EVOL*, 2011.

[110] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[111] W. F. Opdyke and R. E. Johnson. Refactoring: an aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the Symposium on Object Oriented Programming Emphasizing Practical Applications (SOOPPA)*, 1990.

[112] J. L. Overbey, S. Negara, and R. E. Johnson. Refactoring and the evolution of Fortran. In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE, 2009.

[113] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.

[114] H. T. T. Petteri Sevon and P. Onkamo. Gene mapping by pattern discovery. In *Data Mining in Bioinformatics*, 2005.

[115] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *ICSM*, 2010.

[116] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. BugCache for inspections: hit or miss? In *ESEC/FSE*, 2011.

[117] J. Ratzinger, T. Sigmund, P. Vorburger, and H. Gall. Mining software evolution to predict refactoring. In *ESEM*, 2007.

[118] R. Robbes. *Of Change and Software*. PhD thesis, University of Lugano, 2008.

[119] R. Robbes and M. Lanza. A change-based approach to software evolution. *ENTCS*, 166:93–109, January 2007.

[120] R. Robbes and M. Lanza. SpyWare: a change-aware development toolset. In *ICSE*, 2008.

[121] R. Robbes, M. Lanza, and M. Lungu. An approach to software evolution based on semantic change. In *FASE*, 2007.

[122] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 1997.

[123] S. Roock and A. Havenstein. Refactoring tags for automatic refactoring of framework dependent applications. In *Proceedings of Extreme Programming Conference*, XP, 2002.

[124] P. Sevon, H. Toivonen, and V. Ollikainen. TreeDT: Tree pattern mining for gene mapping. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3, 2006.

[125] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR*, 2005.

[126] W. Snipes, B. P. Robinson, and E. R. Murphy-Hill. Code hot spot: A tool for extraction and analysis of code change history. In *ICSM*, 2011.

[127] Apache Subversion centralized version control. http://subversion.apache.org/.

[128] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta. An empirical study on the maintenance of source code clones. In *Empirical Software Engineering*, 2010.

[129] H. Toivonen, P. Onkamo, P. Hintsanen, E. Terzi, and P. Sevon. Data mining for gene mapping. In *Next Generation of Data Mining Applications*, 2005.

[130] G. Uddin, B. Dagenais, and M. P. Robillard. Analyzing temporal api usage patterns. In *ASE*, 2011.

[131] M. Vakilian, N. Chen, R. Z. Moghaddam, S. Negara, and R. E. Johnson. A compositional paradigm of automating refactorings. In *ECOOP*, 2013.

[132] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, 2012.

[133] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The need for richer refactoring usage data. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, PLATEAU, 2011.

[134] F. Van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *IWPSE*, 2003.

[135] F. Van Rysselberghe, M. Rieger, and S. Demeyer. Detecting move operations in versioning information. In *CSMR*, 2006.

[136] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*. Spinger-Verlag, 2004.

[137] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE*, 2006.

[138] Z. Xing and E. Stroulia. Analyzing the evolutionary history of the logical design of object-oriented software. *TSE*, 31:850–868, 2005.

[139] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE*, 2005.

[140] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported - an eclipse case study. In *ICSM*, 2006.

[141] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30, 2004.

[142] Y. Yoon and B. A. Myers. Capturing and analyzing low-level events from the code editor. In *PLATEAU*, 2011.

[143] A. Zaidman, B. V. Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *ICST*, 2008.

[144] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 2000.

[145] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *SDM*, 2002.

[146] M. V. Zelkowitz. Perspectives in software engineering. *ACM Comput. Surv.*, 10, 1978.

[147] G. Zheng, S. Negara, C. L. Mendes, E. R. Rodrigues, and L. V. Kalé. Automatic handling of global variables for multi-threaded MPI programs. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems*, ICPADS, 2011.

[148] T. Zimmermann, N. Nagappan, and A. Zeller. Predicting bugs from history. *Software Evolution*, 2008.

[149] T. Zimmermann, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *ICSE*, 2004.

[150] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31, 2005.