Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?

Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
{snegara2,mvakili2,nchen,rjohnson,dig}@illinois.edu

Abstract. Researchers use file-based Version Control System (VCS) as the primary source of code evolution data. VCSs are widely used by developers, thus, researchers get easy access to historical data of many projects. Although it is convenient, research based on VCS data is incomplete and imprecise. Moreover, answering questions that correlate code changes with other activities (e.g., test runs, refactoring) is impossible. Our tool, CodingTracker, non-intrusively records fine-grained and diverse data *during* code development. CodingTracker collected data from 24 developers: 1,652 hours of development, 23,002 committed files, and 314,085 testcase runs.
This allows us to answer: How much code evolution data is not stored in VCS? How much do developers intersperse refactorings and edits in the same commit? How frequently do developers fix failing tests by changing the test itself? How many changes are committed to VCS without being tested? What is the temporal and spacial locality of changes?

1 Introduction

Any successful software system continuously evolves in response to ever-changing requirements [35]. Developers regularly add new or adjust existing features, fix bugs, tune performance, etc. Software evolution research extracts the code evolution information from the system's historical data. 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 [20], SVN [47], CVS [7]). Therefore, software evolution researchers [1,10,11,13,14,16–19,21,22, 24,27,28,31,34,40,45,46,49–51,54] use VCS to easily 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 a Version Control System (VCS). Therefore, code evolution research performed on the snapshots stored in the VCS (like in [16–18]) 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 [10, 11, 21, 49, 50], infer bug fixes [23, 30, 32, 33, 39, 53]).

Second, VCS data is *imprecise*. A single VCS commit may contain several overlapping changes to the same program entity. For example, a refactored program entity could also be edited in the same commit. 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, refactoring) 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. 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?

Code evolution research studies how the code is changed. So, it is natural to make changes be first-class citizens [42,44] and leverage the capabilities of an Integrated Development Environment (IDE) to capture code changes online rather than trying to infer them post-mortem from the snapshots stored in VCS. We developed a tool, CodingTracker, an Eclipse plug-in that unintrusively collects the 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 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. To represent the raw code edits collected by CodingTracker uniformly and consistently, we implemented an algorithm that infers changes as Abstract Syntax Tree (AST) node operations. Section 2.1 presents more details about our choice of the unit of code change.

We deployed CodingTracker to collect evolution data for 24 developers working in their natural settings. So far, we have 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:

Q1: How much code evolution data is not stored in VCS?
Q2: How much do developers intersperse refactorings and edits in the same commit?
Q3: How frequently do developers fix failing tests by changing the test itself?
Q4: How many changes are committed to VCS without being tested?
Q5: 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 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 [8, 9, 36]. In addition, 24% of changes committed to VCS are untested. This shows the usefulness of continuous integration tools [2, 3, 25, 26]. 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 [4] that allow developers to focus on a particular part of the system.

This paper makes the following major contributions:

1. The design of five questions about the reliability of VCS data in studying code evolution. These five research questions have never been answered before.
2. A field study of 24 Java developers working in their natural environment. To the best of our knowledge, this is the first study to present quantitative evidence of the limitations of code evolution research based on VCS data.
3. CodingTracker, an Eclipse plug-in that collects a variety of code evolution data online and a replayer that reconstructs the underlying code base at any given point in time. CodingTracker is open source and available at http://codingspectator.cs.illinois.edu.
4. A novel algorithm that infers AST node operations from low level code edits.

2 Research Methodology

To answer our research questions, we conducted a user study on 24 participants. We 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. We also recruited 11 programmers who worked on open source projects in different domains, including marketing, banking, business process management, and database management. Table 1 shows the programming experience of our participants[1]. In the course of our study, we collected code evolution data for 1,652 hours of code development with a mean distribution of 69 hours per programmer and a standard deviation of 62.

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 the high-level events like automated refactoring invocations, test runs, and interactions with Version Control System (VCS). CodingTracker employed CodingSpectator's infrastructure [48] to regularly upload the collected data to our centralized repository. Section 4 presents more details about the data CodingTracker collects.

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

Table 1. Programming experience of the participants.

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

2.1 Unit of Code Change

Our code evolution questions require measuring the number of code changes. Therefore, we need to define a unit of code change. 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 our code evolution analysis. A single code edit could represent typing a single character or inserting a whole class declaration. Moreover, even if several code edits 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.

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). We represent a *move* operation, which moves a child node from one parent to another one in an AST, as two consequent AST node operations: *delete* and *add*.

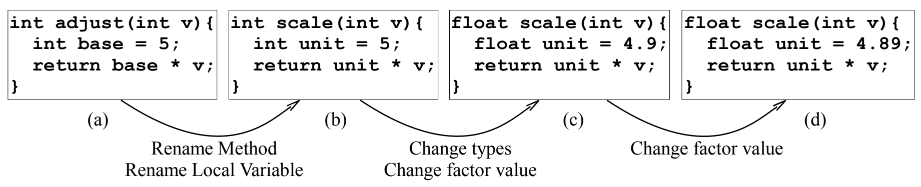
To infer AST node operations from the collected raw edits, we apply our novel inferencing algorithm described in Section 5. Our research questions require establishing how AST node operations correlate with different developer's actions, e.g., whether an AST operation is a result of a refactoring, whether AST operations are followed by a commit or preceded by tests, etc. Therefore, CodingTracker inserts the inferred AST node 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.

3 Research Questions

3.1 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 1 presents a code evolution scenario. The developer checks out the latest revision of the code shown in Figure 1(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 1(d) into the VCS.



```
int adjust(int v){
  int base = 5;
  return base * v;
}
```

```
int scale(int v){
  int unit = 5;
  return unit * v;
}
```

```
float scale(int v){
  float unit = 4.9;
  return unit * v;
}
```

```
float scale(int v){
  float unit = 4.89;
  return unit * v;
}
```

Fig. 1. A code evolution scenario that illustrates a shadowed code change and an overlap of refactorings with other code changes.

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 1(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 in 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 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 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 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 3 presents the results. Overall, 78% of shad-

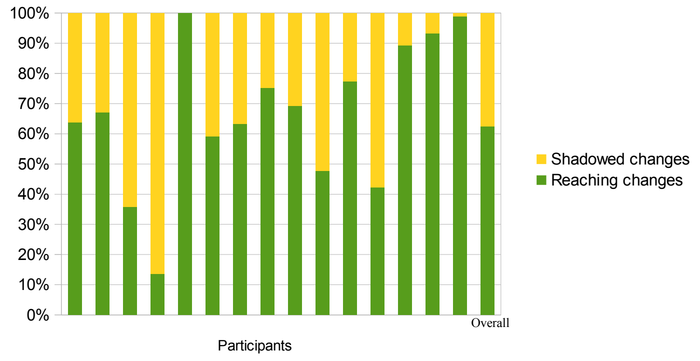


Fig. 2. Ratio of reaching and shadowed changes.

owed 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.

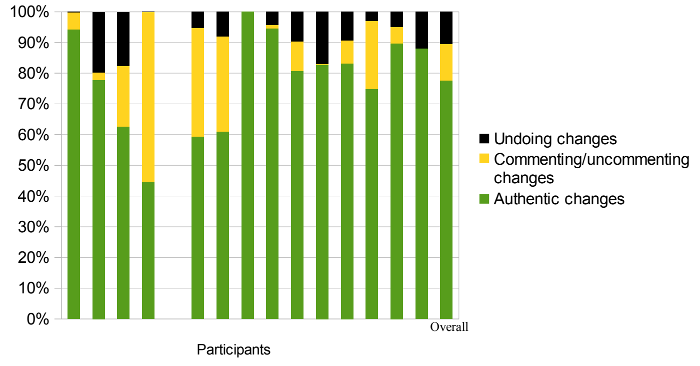


Fig. 3. Composition of shadowed changes. The fifth bar is missing, since there are no shadowed changes for this participant.

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. Further research is required to investigate the extent of this imprecision.

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

Many tools [10,11,21,29,49,50] 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 [11]. If a refactored program entity is additionally 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 1 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 1 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 for ten participants who both used Eclipse-based VCS clients and performed automated refactorings. Note that we do not consider manual refactorings since they can not 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 4 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 4 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. We consider a program entity refactored/edited if any of the entities of its cluster is affected by an automated refactoring/manual edit. Figure 5 demonstrates that on average, 46% of program entities are both refactored and additionally changed in the same commit.

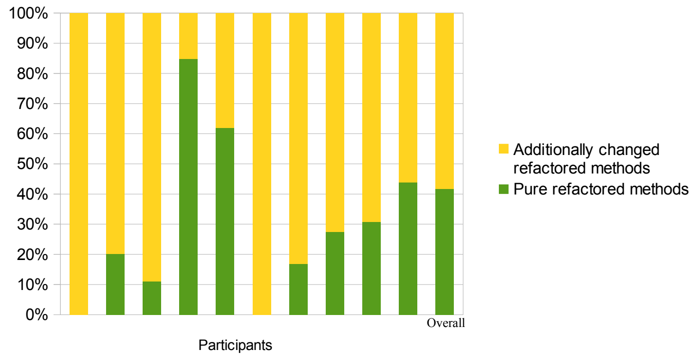


Fig. 4. Ratio of purely refactored methods and those that are both refactored and additionally changed before being committed to VCS.

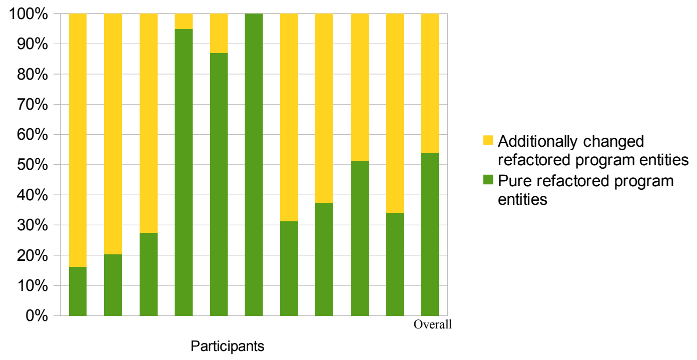


Fig. 5. Ratio of purely refactored program entities and those that are both refactored and additionally changed before reaching a commit.

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 [10,11,21,29,49,50]. Our findings serve as a strong motivation to build a refactoring inference tool based on the precise, fine-grained data collected by CodingTracker and compare its accuracy against the existing snapshot-based tools.

3.3 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 [8, 9, 36].

Figure 6 presents a unit test of a parser. This test checks that the parser produces a specific number of elements 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.

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

Fig. 6. A unit test of a parser that checks the total number of elements in the parser's result.

We justify the need for the 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, 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 the test's package during this time span, we consider that fixing this failing test involves changing it.

Figure 7 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 [9] could have benefited all of the participants, potentially helping to fix more than one third of all failing tests. Nevertheless, only a designated study would show how much of the required changes to the failing tests could be automated by ReAssert.

3.4 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 to break the build and consequently, cause the disruption of the development process. To prevent such scenarios and catch broken builds early, the industry adopted continuous integration tools (e.g., Apache Gump [2], Bamboo [3], Hudson [25], and Jenkins [26]), 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 pervasive yet. In particular, the majority of the projects that we studied

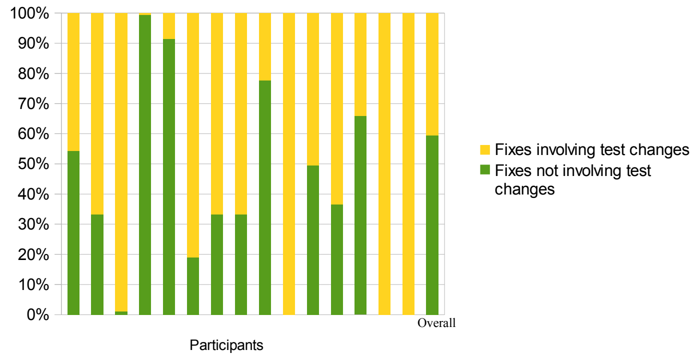


Fig. 7. Ratio of test fixes involving and not involving changes to the tests.

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 in 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 two consecutive commits of a source code file. If there are no test runs in between these two commits, we disregard this pair of commits[2]. Otherwise, we count the total number of code changes that happen in 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 in between the two commits, we get the *tested* changes.

Figure 8 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 9 shows that 97% of the untested changes are authentic, i.e., we discard undos and comments.

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.

[2] We are conservative in calculating the amount 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).

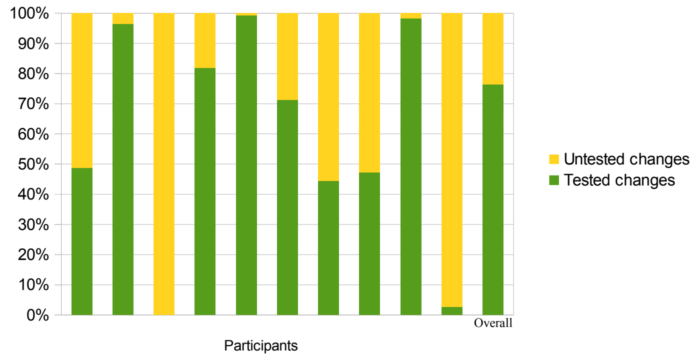


Fig. 8. Ratio of tested and untested code changes that reach VCS.

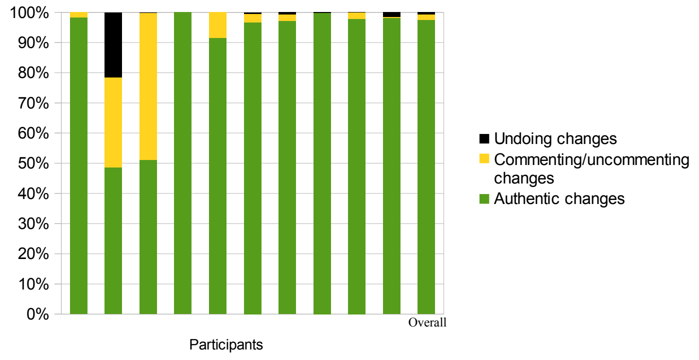


Fig. 9. Composition of untested changes that reach VCS.

3.5 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 [4] 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 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 10 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.

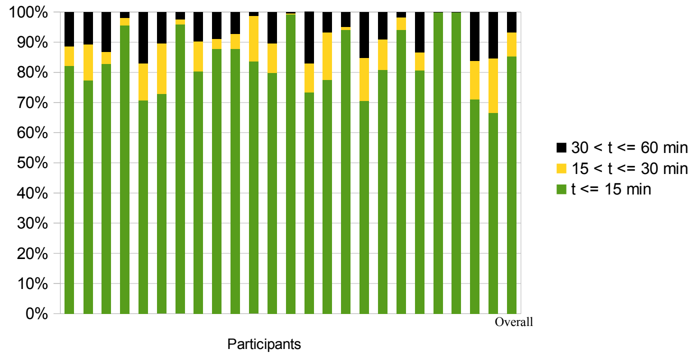


Fig. 10. Ratio of method-level code changes for three sliding time windows: 15, 30, and 60 minutes.

4 Collecting Code Evolution Data

To collect code evolution data for our research questions, we developed an Eclipse plug-in, CodingTracker. CodingTracker registers 38 different kinds of code evolution events that are grouped in ten categories. Table 2 presents the complete list of the registered events. CodingTracker records the detailed information about each registered event, including the timestamp at which the 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. Note that we need to replay the recorded data to reproduce the actions of a developer since our AST node operations inferencing algorithm is applied offline.

CodingTracker's replayer is an Eclipse View that is displayed alongside other Views of an Eclipse workbench, thus enabling a user to see the results of the replayed events in the same Eclipse instance. The replayer allows to load a recorded sequence of events, browse it, hide events of the kinds that a user is not interested in, and replay the sequence at any desired pace.

Table 2. The complete list of events recorded by CodingTracker.

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

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[3].

Eclipse creates a refactoring descriptor for every performed automated refactoring. 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, can not 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 [5]. Therefore, 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 *Text editing*, *File editing*, and *Resource manipulation* categories presented in Table 2. In a sequence of recorded events, effects of an automated refactoring are located in between its *Start refactoring* and *Finish refactoring* events. To ensure robust replaying, CodingTracker replays the recorded refactorings' effects rather than their descriptors.

5 AST Node Operations Inferencing Algorithm

Our inferencing algorithm converts the raw text edits collected by CodingTracker 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.

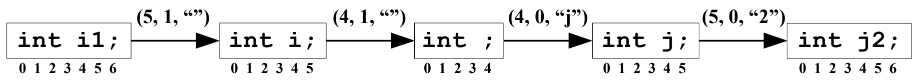
Given an edited document, a single text edit is fully described by a 3-tuple `(<offset>, <removed text length>, <added text>)`, where `<offset>` is the offset of the edit in the edited document, `<removed text length>` is the length of the text that is removed at the specified offset, and `<added text>` is the text that is added at the specified offset. If `<removed text length>` is 0, the edit

[3] Note that replaying CVS/SVN commits does not involve any interactions with Version Control System (VCS), but rather checks the correctness of the replaying process.

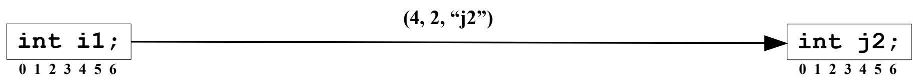
does not remove any text. If `<added text>` is empty, the edit does not add any text. If `<removed text length>` is not 0 and `<added text>` is not empty, the edit replaces the removed text with the `<added text>` in the edited document.

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

Gluing Figure 11(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 four 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 11(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 intentions of a developer.



(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 11(a).

Fig. 11. An example of changing a variable's name represented both as individual text edits recorded by CodingTracker (Figure 11(a)) and as a single *coherent text edit* (Figure 11(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.

To decide whether a text edit `e2` should be "glued" to a preceding text edit `e1`, we use the following heuristics:

1. `e2` should immediately follow `e1`, i.e., there are no other events in between these two text edits.
2. `e2` should continue the text change of `e1`, i.e., text edit `e2` should start 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.

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 Our AST node operations inferencing algorithm processes coherent text edits as soon as they are constructed, except the cases when text edits introduce parse errors in the underlying code. Parse 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, we postpone the inferencing until the AST is well-formed again. Such postponing causes accumulation of several coherent text edits, which are processed by our inferencing algorithm together. Figure 12 shows an example of a 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 we believe that such scenarios are very rare in practice. In particular, our personal experience of replaying the recorded sequences shows that the code rarely remains in the unparsable state for long time.

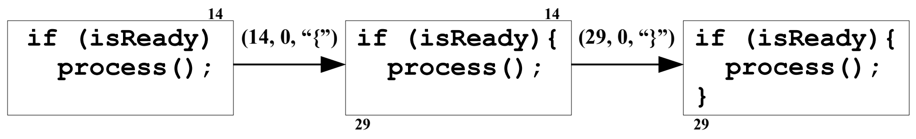


Fig. 12. An example of two code edits: the first edit breaks the AST of the edited program, while the second edit brings the AST back to a well-formed state.

Pseudocode Figure 13 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 at that point, 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 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, every descendant node of the common covering node in the old AST is checked against the edits in *cteList* (lines 6 – 18). 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 12). If no edits affect a node, the algorithm looks for its matching node in the new AST (line 15). Every matched pair of nodes is added to *matchedNodes*.

In the following step (lines 19 – 25), the inferencing algorithm matches yet unmatched nodes that have the same AST node types and the same position in the old and the new ASTs. Finally, the algorithm creates an *update* operation for every matched node whose content has changed (lines 26 – 30), a *delete*

```
input: $oldAST$, $newAST$, $cteList$ // the list of coherent text edits
output: $astNodeOperations$
1  $astNodeOperations = \emptyset$;
2  $matchedNodes = \emptyset$;
3  $coveringPosition$ = getCommonCoveringNodePosition($oldAST$, $newAST$, $cteList$);
4  $oldCoveringNode$ = getNode($oldAST$, $coveringPosition$);
5  $newCoveringNode$ = getNode($newAST$, $coveringPosition$);
6  foreach ($oldNode \in$ getDescendants($oldCoveringNode$)) { // matches outliers
7    $deltaOffset = 0$;
8    foreach ($textEdit \in cteList$) {
9      if (affects($textEdit$, $oldNode$, $deltaOffset$) {
10         continue foreach_line6;
11     } else {
12         $deltaOffset$ += getEditOffset($textEdit$, $oldNode$, $deltaOffset$);
13     }
14   }
15   if ($\exists$ $newNode \in$ getDescendants($newCoveringNode$) :
          getOffset($oldNode$) + $deltaOffset$ == getOffset($newNode$) &&
          haveSameASTNodeTypes($oldNode$, $newNode$)) {
16     $matchedNodes \cup= (oldNode, newNode)$;
17   }
18 }
19 foreach ($oldNode \in$ getDescendants($oldCoveringNode$) :
            $oldNode \notin$ getOldNodes($matchedNodes$)) { // matches same-position nodes
20   $oldPosition$ = getNodePositionInAST($oldNode$, $oldAST$);
21   $newNode$ = getNode($newAST$, $oldPosition$);
22   if ($\exists$ $newNode \in$ getDescendants($newCoveringNode$) :
                                      haveSameASTNodeTypes($oldNode$, $newNode$) {
23     $matchedNodes \cup= (oldNode, newNode)$;
24   }
25 }
26 foreach (($oldNode$, $newNode$) $\in matchedNodes$) {
27   if (getText($oldNode$) $\neq$ getText($newNode$)) {
28     $astNodeOperations$ $\cup=$ getUpdateOperation($oldNode$, $newNode$);
29   }
30 }
31 foreach ($oldNode \in$ getDescendants($oldCoveringNode$) :
                                        $oldNode \notin$ getOldNodes($matchedNodes$)) {
32   $astNodeOperations$ $\cup=$ getDeleteOperation($oldNode$);
33 }
34 foreach ($newNode \in$ getDescendants($newCoveringNode$) :
                                        $newNode \notin$ getNewNodes($matchedNodes$)) {
35   $astNodeOperations$ $\cup=$ getAddOperation($newNode$);
36 }
```

Fig. 13. Overview of our AST node operations inferencing algorithm.

operation for every unmatched node in the old AST (lines 31 – 33), and an *add* operation for every unmatched node in the new AST (lines 34 – 36).

Example Figure 14 illustrates a coherent text edit that changes a variable declaration. Figure 15 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.

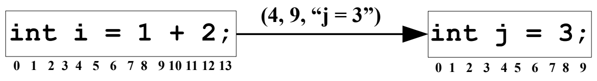


Fig. 14. An example of a text edit that changes a variable declaration.

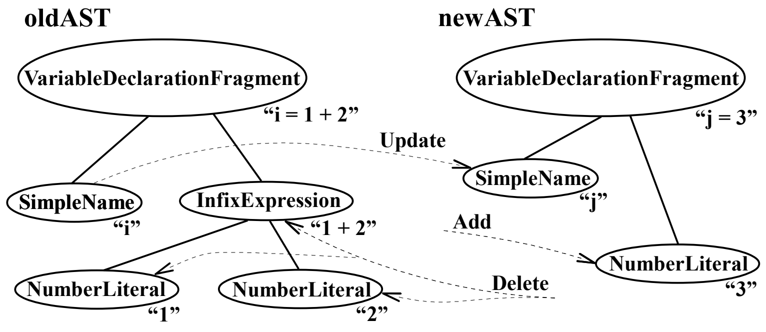


Fig. 15. The inferred AST node operations for the text edit in Figure 14.

6 Threats to Validity

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

6.1 Experimental Setup

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

Many factors affect programmers' practices. For example, programmers 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 programmers 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). Therefore, those participants who used CodingTracker more influenced our results more.

Our results are limited to programmers 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.

6.2 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, as any heuristics, they can not 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, such corner case scenarios are infrequent and thus, their influence on our results is negligible.

The current implementation of our AST node operations inferencing algorithm does not support the *move* operation, but rather represents 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 states could cause the algorithm to produce some noise in the form of spurious or missing AST node operations. Such scenarios are very uncommon and hence, their impact on our results is minimal.

7 Related Work

7.1 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. [13] identified specific indicators for *code decay* by conducting a study on a large (∼100,000,000 LOC) real time software 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. [51] analyzed the evolution of design in object-oriented software by reconstructing the 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. [16] 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 [41, 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.

7.2 Tools for Reconstructing Program 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. [15] proposed an 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. [29] 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 could automatically infer those change rules and present them as concise and understandable transformations to the programmer.

Weissgerber et al. [50] and Dig et al. [11] proposed tools for identifying refactorings between two different version of a source code. Such tools help developers gain better insights into the high-level transformations that occurred between different versions of a program.

All these tools detect structural changes in the evolving code using VCS snapshots. However, the results of our field study presented in Section 3 show that VCS snapshots provide incomplete and imprecise data, thus compromising the accuracy of these tools. The accuracy of such tools could be greatly improved by working on the fine-grained changes provided through a change-based software tool such as CodingTracker.

7.3 Tools for Fine-grained Analysis of Code Evolution

Robbes et al. [42, 44] proposed to make a change the first-class citizen and capture it directly from an IDE as soon as it happens. They developed a tool, SpyWare [43], that implements these ideas. SpyWare gets notified by the Smalltalk compiler in the 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. [12] 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 field and method declarations and up.

Omori and Maruyama [37, 38] 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. [52] 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 software evolution 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. [6] 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.

8 Conclusions

The primary source of data in code evolution research is the file-based Version Control System (VCS). Our results 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 conducted a field study using CodingTracker, our Eclipse plug-in, that collects diverse evolution data. We analyzed the collected data and answered five code evolution research questions. 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.

Acknowledgments. This work 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.

References

1. Adams, B., Jiang, Z.M., Hassan, A.E.: Identifying crosscutting concerns using historical code changes. In: ICSE (2010)

2. Apache Gump continuous integration tool. http://gump.apache.org/
3. Bamboo continuous integration and release management. http://www.atlassian.com/software/bamboo/
4. Bragdon, A., Reiss, S.P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., LaViola, Jr., J.J.: Code Bubbles: rethinking the user interface paradigm of integrated development environments. In: ICSE (2010)
5. Eclipse bug report. https://bugs.eclipse.org/bugs/show_bug.cgi?id=365233
6. Chan, J., Chu, A., Baniassad, E.: 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)
7. CVS - Concurrent Versions System. http://cvs.nongnu.org/
8. Daniel, B., Gvero, T., Marinov, D.: On test repair using symbolic execution. In: ISSTA (2010)
9. Daniel, B., Jagannath, V., Dig, D., Marinov, D.: ReAssert: Suggesting repairs for broken unit tests. In: ASE (2009)
10. Demeyer, S., Ducasse, S., Nierstrasz, O.: Finding refactorings via change metrics. In: OOPSLA (2000)
11. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.E.: Automated detection of refactorings in evolving components. In: ECOOP (2006)
12. EclipsEye. http://www.inf.usi.ch/faculty/lanza/Downloads/Shar07a.pdf
13. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does code decay? assessing the evidence from change management data. TSE 27, 1–12 (January 2001)
14. Eshkevari, L.M., Arnaoudova, V., Di Penta, M., Oliveto, R., Guéhéneuc, Y.G., Antoniol, G.: An exploratory study of identifier renamings. In: MSR (2011)
15. Fluri, B., Wuersch, M., Pinzger, M., Gall, H.: Change distilling: Tree differencing for fine-grained source code change extraction. TSE 33, 725–743 (November 2007)
16. Gall, H., Hajek, K., Jazayeri, M.: Detection of logical coupling based on product release history. In: ICSM (1998)
17. Gall, H., Jazayeri, M., Klsch, R.R., Trausmuth, G.: Software evolution observations based on product release history. In: ICSM (1997)
18. Gall, H., Jazayeri, M., Krajewski, J.: CVS release history data for detecting logical couplings. In: IWMPSE (2003)
19. Girba, T., Ducasse, S., Lanza, M.: Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In: ICSM (2004)
20. Git - the fast version control system. http://git-scm.com/
21. Gorg, C., Weisgerber, P.: Detecting and visualizing refactorings from software archives. In: ICPC (2005)
22. Hassaine, S., Boughanmi, F., Guéhéneuc, Y.G., Hamel, S., Antoniol, G.: A seismology-inspired approach to study change propagation. In: ICSM (2011)
23. Hassan, A.E.: Predicting faults using the complexity of code changes. In: ICSE (2009)
24. Hindle, A., German, D.M., Holt, R.: What do large commits tell us?: a taxonomical study of large commits. In: MSR (2008)
25. Hudson extensive continuous integration server. http://hudson-ci.org/
26. Jenkins extendable open source continuous integration server. http://jenkins-ci.org/
27. Kagdi, H., Collard, M.L., Maletic, J.I.: A survey and taxonomy of approaches for mining software repositories in the context of software evolution. J. Softw. Maint. Evol. 19 (March 2007)

28. Kawrykow, D., Robillard, M.P.: Non-essential changes in version histories. In: ICSE (2011)
29. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE (2007)
30. Kim, S., Jr., E.J.W., Zhang, Y.: Classifying software changes: Clean or buggy? TSE 34(2) (2008)
31. Kim, S., Pan, K., Whitehead, Jr., E.J.: Micro pattern evolution. In: MSR (2006)
32. Kim, S., Zimmermann, T., Pan, K., Whitehead, E.J.J.: Automatic identification of bug-introducing changes. In: ASE (2006)
33. Lee, T., Nam, J., Han, D., Kim, S., In, H.P.: Micro interaction metrics for defect prediction. In: ESEC/FSE (2011)
34. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc. (1985)
35. Lehman, M.M.: Programs, life cycles, and laws of software evolution. Proc. IEEE 68(9), 1060–1076 (September 1980)
36. Mirzaaghaei, M., Pastore, F., Pezze, M.: Automatically repairing test cases for evolving method declarations. In: ICSM (2010)
37. Omori, T., Maruyama, K.: A change-aware development environment by recording editing operations of source code. In: MSR (2008)
38. Omori, T., Maruyama, K.: An editing-operation replayer with highlights supporting investigation of program modifications. In: IWMPSE-EVOL (2011)
39. Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P.: BugCache for inspections: hit or miss? In: ESEC/FSE (2011)
40. Ratzinger, J., Sigmund, T., Vorburger, P., Gall, H.: Mining software evolution to predict refactoring. In: ESEM (2007)
41. Robbes, R.: Of Change and Software. Ph.D. thesis, University of Lugano (2008)
42. Robbes, R., Lanza, M.: A change-based approach to software evolution. ENTCS 166, 93–109 (January 2007)
43. Robbes, R., Lanza, M.: SpyWare: a change-aware development toolset. In: ICSE (2008)
44. Robbes, R., Lanza, M., Lungu, M.: An approach to software evolution based on semantic change. In: FASE (2007)
45. Śliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: MSR (2005)
46. Snipes, W., Robinson, B.P., Murphy-Hill, E.R.: Code hot spot: A tool for extraction and analysis of code change history. In: ICSM (2011)
47. Apache Subversion centralized version control. http://subversion.apache.org/
48. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., Johnson, R.E.: Use, disuse, and misuse of automated refactorings. In: ICSE (2012)
49. Van Rysselberghe, F., Rieger, M., Demeyer, S.: Detecting move operations in versioning information. In: CSMR (2006)
50. Weissgerber, P., Diehl, S.: Identifying refactorings from source-code changes. In: ASE (2006)
51. Xing, Z., Stroulia, E.: Analyzing the evolutionary history of the logical design of object-oriented software. TSE 31, 850–868 (October 2005)
52. Yoon, Y., Myers, B.A.: Capturing and analyzing low-level events from the code editor. In: PLATEAU (2011)
53. Zimmermann, T., Nagappan, N., Zeller, A.: Predicting bugs from history. Software Evolution (2008)
54. Zimmermann, T., Weisgerber, P., Diehl, S., Zeller, A.: Mining version histories to guide software changes. In: ICSE (2004)