Enhancing Software Development Techniques via Copy Codebases

Kıvanç Muşlu

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Michael D. Ernst, Chair

Yuriy Brun

Andrew J. Ko

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

Abstract

Enhancing Software Development Techniques via Copy Codebases

Kıvanç Muşlu

Chair of the Supervisory Committee:
Professor Michael D. Ernst
Computer Science & Engineering

Most software development tasks require developers to interact with multiple versions of a codebase directly or through an analysis tool. Consider a developer, Alice, who wants to implement a new feature. To ensure that her changes are of high quality, Alice might want to continuously analyze the current codebase. To make the changes fast but with confidence, Alice might use automated transformations such as refactorings. While using these transformations Alice might want to analyze the likely future codebases, which these transformations would generate, to make more informed decisions. Finally, to pinpoint the cause of any regression defect, Alice might want to analyze historical codebases e.g., to binary search the development history.

Unfortunately, developers' interaction with multiple codebases is limited since modern integrated development environments (IDEs) maintain one version of the codebase with limited development history. First, it is difficult to run an arbitrary analysis continuously with development since most analyses assume that the code under analysis does not change for the duration of the analysis. Second, having access to one codebase makes it impossible to analyze likely future codebases continuously with development since the modifications done to generate the future codebases conflict with development. Third, maintaining a limited development history makes it difficult to analyze historical codebases e.g., to extract information from the development history.

This dissertation introduces a novel technique, Codebase Replication, which creates and incrementally maintains a copy of the developer's codebase. Our thesis is that having access to copy codebases improves developers' interaction with the current codebase, likely future codebases, and historical codebases. Improving the developer's interaction with codebases will simplify software engineering tasks and reduce developer mistakes.

Continuous analyses — analyses that run in the background after each developer edit and update their result unobtrusively — improve developers' interaction with the current codebase. Having access to an in-sync copy codebase simplifies the design and implementation of continuous analyses. We introduce Codebase Analysis, which lets developers run an existing analysis continuously with development without worrying about conflicting edits. We prototyped Solstice, an implementation of Codebase Analysis for the Eclipse IDE. Solstice has negligible (< 2.5 ms) synchronization delay and IDE overhead. Using Solstice, we implemented four continuous analyses. Each of these analyses required less than 800 lines of Java code and 20 hours of development time, on average. Our case study with Solstice continuous testing shows that developers like continuous feedback and they like continuous analysis tools built with Solstice.

Impure analyses — analyses that modify the source code before computing results — improve developers' interaction with likely future codebases. We show how Codebase Analysis supports impure analyses and guarantees that these analyses' modifications will not conflict with development. We introduce a new impure analysis that augments IDE recommendations with their speculatively computed consequences. We created a prototype implementation, Quick Fix Scout, in the context of Eclipse Quick Fixes. Our experiment shows that Quick Fix Scout can speed up compilation error removal tasks by 10%.

Being able to access the development history in the granularity that is the most optimal for the underlying software engineering task improves developers' interaction with historical codebases. We introduce Codebase Manipulation, a history manipulation framework that lets developers access the history at different granularities for different tasks. We prototyped Bread, an implementation of Codebase Manipulation for the Eclipse IDE. Bread automatically creates a fine-grained development history by recording each edit. We show that Codebase Manipulation can express a range of development tasks by providing algorithms for three common information-retrieval-based tasks. Using Bread, we prototyped two of these algorithms. Our initial experience with these tools suggests that Bread can simplify information-retrieval-based software engineering tasks.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my advisers, Michael D. Ernst, Yuriy Brun, and David Notkin. Their help and support made it possible to identify and unify the research topics in this dissertation. Ezgi Mercan, Erkan Keremoğlu and Safiye Çelik provided feedback on an early draft of this dissertation. Sai Zhang provided references for related work on Untangler and Deniz Altınbüken provided references on how distributed systems research uses replication. Reid Holmes is a collaborator for the Speculative Analysis chapter and Luke Swart is a collaborator for the Codebase Manipulation chapter.

DEDICATION

to my beloved parents Erkan and Şükriye, and to my lovely wife and eternal soulmate Yağmur.

Chapter 1
INTRODUCTION

A codebase is a collection of resources that are required to build a software system, including the source code and the data dependencies. Software development tasks require the developer to interact with different versions of the codebase. Consider a developer, Alice, who wants to implement a new feature. While implementing this feature, Alice would prefer if the recent changes (1) are of high quality, (2) are implemented fast but with confidence, and (3) do not introduce defects. To ensure that the recent changes are of high quality, Alice might want to run continuous program analyses, such as static defect finding tools. A continuous analysis runs in the background after each developer edit and updates its results unobstrusively. To ensure that the recent changes are implemented fast but with confidence, Alice might use automated transformations such as refactorings, auto completions, and quick fixes. While using these transformations, Alice might want to run speculative analyses to make more informed decisions. A speculative analysis computes the consequences of an action before that action is executed. Finally, to pinpoint the cause of any regression defect introduced by the recent changes, Alice might want to binary search the development history. During this task, Alice interacts with the following codebases: current (to run program analyses), likely future (to run speculative analyses), and historical (to search the development history).

Developers' interaction with the current codebase is well-studied. Modern integrated development environments (IDEs), such as Eclipse [39] and Visual Studio [147], support code navigation through hyperlinks and a browsable navigation history. Code Bubbles [15, 16] and Code Canvas [36] improve code understanding and navigation by letting the developer focus on code pieces and the relation between them. Mylar [89, 127] improves code navigation by letting the developer save and switch between development contexts [90]. Additionally, modern IDEs make it easier and more robust to modify the current codebase through simple code transformations, such as refactorings, auto completions, and quick fixes. Code Recommenders [31] uses past developer actions and the current context to reorder and filter code completion proposals [18, 125]. BeneFactor [58, 57] and WitchDoctor [56] infer an ongoing manual refactoring and complete it with an automated one.

Unfortunately, software development tasks require more than navigating the current codebase and simple code transformations. Developers might need to interact directly or indirectly — through a program analysis — with the current, likely future, and historical codebases. Some of this interaction is more useful when it is continuous. For example, a continuous program analysis causes fewer distractions and ensures that the analysis results are always available with the least possible delay. For the above-mentioned example, Alice wanted to analyze (1) the current codebase continuously with development, (2) the likely future codebases continuously with development, and (3) the historical codebases.

Having access to a single codebase limits software development. The complexity of running an arbitrary analysis continuously with development limits developers' interaction with the current codebase. For some analyses, such as testing [78, 115] and compilation [45], developers can use an existing continuous implementations, however building continuous analyses in general is difficult since developers' edits conflict with the ongoing analysis execution. The complexity of designing and building continuous speculative analyses limits developers' interaction with the likely future codebases. A continuous speculative analysis cannot work on the developer's codebase since the changes done by the analysis would confuse the developer. Finally, recording a limited development history at a fixed granularity limits developers' interaction with the historical codebases since different development tasks require the developer to access the development history in different granularities.

This dissertation reduces or removes the above-mentioned limitations. Our thesis is that software development can be enhanced by improving developers' interaction with (1) the current codebase through continuous analysis feedback, (2) likely future codebases through speculative analysis of likely developer actions, which helps the developer make more informed decisions, and (3) historical codebases by maintaining a complete and fine-grained history, and letting the developer explore this history in the best granularity for the underlying task. All enhancements build upon a powerful foundation: an incrementally-maintained copy of the developer's codebase can be used to run arbitrary computation — including program analyses — and store additional information, in parallel with development.

The rest of the dissertation is organized as follows: Chapter 2 introduces a novel technique, Codebase Replication, that creates and incrementally maintains a copy of the developer's codebase in real time. Chapters 3, 4, and 5 extend Codebase Replication and introduce three novel approaches that enhance software development. Codebase Analysis makes it easier to design and build continuous analysis. Speculative Analysis explores likely future actions and helps the developer make more informed decisions. Codebase Manipulation maintains a complete and fine-grained development history and simplifies information retrieval from the development history by letting the developer access this history at the best granularity for the underlying task. Chapter 6 puts the dissertation in the context of the related research. Chapter 7 concludes with the contributions.

Chapter 2
CODEBASE REPLICATION

Techniques described in this dissertation (Chapters 3, 4, and 5) manifest as frameworks that analyze, modify, or store data on the developer's codebase. Developer edits conflict with these frameworks since they both access the same codebase, simultaneously. For example, an impure analysis modifies the developer's codebase before computing results. Running an impure analysis on the developer's codebase, simultaneously with development would confuse the developer and become detrimental. We developed Codebase Replication, a novel technique [105, 106] that incrementally maintains a copy of the developer's codebase, in real time. Codebase Replication removes the above-mentioned problem by letting these frameworks run programs and store additional data on an up-to-date copy codebase.

Codebase Replication satisfies two design goals: *isolation* — ensuring that the framework programs do not interrupt the development — and *currency* — ensuring that the framework programs access an up-to-date version of the developer's codebase with minimal delay. *Isolation* is achieved by running the framework programs on the copy codebase, which ensures that their side-effects are contained within the copy codebase and not visible to the developer. *Currency* is achieved by detecting developer edits at the finest granularity and incrementally applying these edits to the copy codebase at times that do not conflict with an ongoing program execution.

Previous approaches that runs programs concurrently with development fail to achieve either isolation or currency. For example, most continuous analysis frameworks, such as Eclipse's Incremental Project Builders [42], achieve some currency by running analyses on the developer's code, however they cannot guarantee isolation. Similarly, integration servers, such as Jenkins [82] and Hudson [77], satisfy isolation by running programs on copy codebases at the integration server, however they fail to satisfy currency since they only update these copies when a special action, such as committing new changes, takes place.

Figure 2.1 illustrates how Codebase Replication, the IDE, and a framework interact with each other. Codebase Replication detects developer actions, including all edits to the codebase, through IDE application programming interface (API). Codebase Replication incrementally maintains a copy codebase that is always in sync with the developer's code by applying all actions that modify the source code to this copy codebase. The frameworks that extend Codebase Replication run their programs or store additional data on this copy codebase, without interrupting development, while still having access to the most recent codebase.

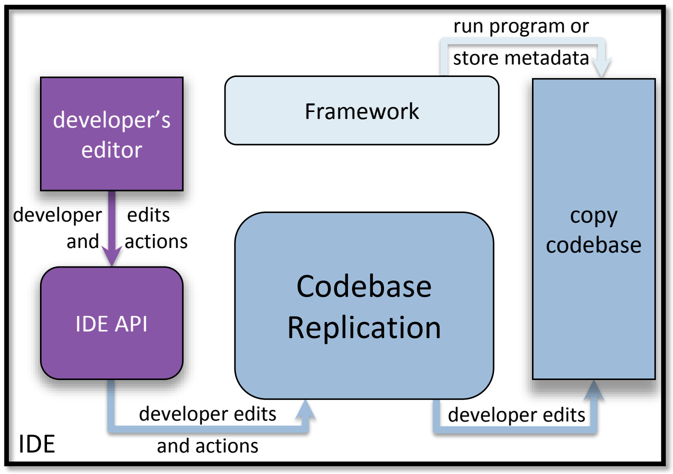


Figure 2.1: Codebase Replication architecture (colored). Codebase Replication (blue) listens to developer actions — including all edits — using IDE application programming interface (purple). Edits are applied to the copy codebase. A framework (light blue) runs a program on the copy codebase while the developer continues working on the main codebase.

Codebase Replication is implemented (together with Codebase Analysis (Chapter 3)) as an Eclipse plug-in, called Solstice [111]. Using Solstice, we showed that Codebase Analysis — hence Codebase Replication — have negligible IDE overhead ($\leq$ 3ms) and high currency during the incremental synchronization of the copy codebase ($\leq$ 3ms) for common developer actions [105].

Chapter 3

CODEBASE ANALYSIS: PROVIDING CONTINUOUS ANALYSIS FEEDBACK

When a developer edits source code, the sooner the developer learns the changes' effects on program analyses, the more helpful those analyses are. A delay can lead to wasted effort or confusion [12, 88, 128]. Ideally, the developer would learn the implications of a change as soon as the change is made.

A few analysis tools already provide immediate feedback. IDEs such as Eclipse and Visual Studio continuously compile the code to inform developers about a compilation error as soon as a code change causes one. Continuous testing informs the developer as soon as possible after a change breaks a test [128]. Speculative conflict detection informs the developer of a conflict soon after the developer commits conflicting changes locally [21]. Speculative quick fix informs the developer of the implications of making a compilation-error-fixing change even *before* the change takes place [109].

Unfortunately, most analysis tools are not designed to continuously provide up-to-date results. Instead, a developer may need to initiate the analysis manually.[1] To ease converting these offline analysis tools into continuous analysis tools [33] that run automatically and always provide up-to-date results, we introduce Codebase Analysis.

After computing analysis results, a continuous analysis tool may indicate that information is available, display the analysis results, or prompt other relevant analyses or tools to run. Some analyses benefit more than others from continuous execution. For example, a fast continuous analysis provides more frequent and earlier feedback than a slow continuous analysis. Even for long-running analyses, continuous execution could provide results to the developer after an external

[1]Only 17% of Eclipse plug-ins listed on GitHub that implement analyses are continuous and reactive to the developer's latest changes; see Section 3.2.1.

interruption, such as receiving a phone call. This is sooner than the developer would otherwise learn the results, and, again, without requiring the developer to initiate the analysis.

Our goal is not to make analyses run faster nor incrementally, but to make it easy to run them more frequently and to simplify the developer's workflow — all without requiring a redesign of the analysis tool. Research that makes analyses run faster, including partial and incremental computation, is orthogonal to our work.

Making an analysis continuous is challenging. This explains why few continuous analyses exist, despite their benefits. Two major challenges are *isolation* and *currency*. Isolation requires that (1) the analysis should not prevent the developer from making new changes, and code changes made by an impure (side-effecting) analysis should not alter the code while the developer is working, and (2) developer edits should not make results of an ongoing analysis potentially stale. Currency requires that (1) analysis results are made available as soon as possible, and (2) results that are outdated by new developer edits are identified as stale.

Building on Codebase Replication, Codebase Analysis addresses these challenges by employing two additional principles:

1. *exclusive ownership* — allowing analyses to request exclusive write access to the copy of the developer's codebase, and
2. *staleness detection* — identifying results made stale by new developer changes.

Existing analysis automation techniques fail to achieve simultaneous isolation and memory-change currency. Build tools such as Ant [1] and Maven [2] can be automated to achieve file-change currency, but they do not achieve isolation since the developer and the analysis work on the same codebase. These build tools could create a separate copy of the codebase and achieve isolation in the same way as integration servers, such as Hudson [77] and SonarQube [138], but such an approach cannot achieve memory-change currency: build tools can only access the code saved to disk, whereas integration servers can only access the latest version control commit. Codebase Analysis enables isolated memory-change-triggered continuous analyses without increasing implementation complexity.

If the developer edits the code while the analysis is running, Codebase Analysis can choose from a variety of reactions: terminate and restart the offline analysis so that the produced results are always accurate; defer propagating the developer's edits to the copy codebase until the analysis is finished so that the analysis can complete, in case the results are useful even when a little stale; or complete the analysis and use analysis-specific logic to mark parts of the results as stale. Since computing analysis results early and often means that more accurate results are available sooner after the developer changes the code, the results could be shown to the developer immediately to reduce wasted time [12, 88, 128], or less frequently to avoid distracting and annoying the developer [13]. Codebase Analysis supports both. An up-to-date analysis result might become stale after an edit to the program. In this case, a continuous analysis can either immediately remove the stale results and indicate that the offline analysis is re-executing on the updated program, or mark the results stale, but keep them until the new results are computed, so that the results are not removed from the developer's context. Codebase Analysis supports both.

If a UI were poorly designed to interrupt the developer, then the continuous analysis results could be distracting to the developer. However, if the information is presented unobtrusively and the developer is permitted to act on it when he or she chooses, then developers find it useful. This has been confirmed experimentally by Saff et al. [129] and is reflected by the popularity of continuous analysis tools such as continuous compilation. Codebase Analysis automatically updates the analysis results in a separate GUI element without disturbing the developer. The developer can make this GUI element (in)visible to ignore or have access to the analysis results.

The main contributions of this chapter are:

- A discussion of the three major design dimensions of continuous analysis implementations (Section 3.2).
- A Codebase Analysis design that addresses currency and isolation (Section 3.3), including two alternatives for adding external interruption support to a continuous analysis implementation to increase its input currency without violating the analysis isolation (Section 3.3.3).
- Solstice, an Eclipse-based realization of Codebase Analysis that brings isolation and currency to offline analyses for easily converting them into continuous analysis Eclipse plug-ins (Section 3.4).

- An evaluation of Solstice's performance, in terms of overhead (isolation cost) and responsiveness to changes (currency) (Section 3.5.1).
- Four publicly-available, continuous analysis plug-ins with isolation and currency properties, and an evaluation of the ease of building such plug-ins with Solstice (Section 3.5.2).
- An evaluation of the Solstice continuous testing plug-in in two case studies, demonstrating that Solstice (and therefore Codebase Analysis) continuous analysis tools are intuitive and easy to use, and are liked by the programmers (Section 3.5.3).

The rest of this chapter is organized as follows: Section 3.1 defines concepts used in the rest of this chapter. Section 3.2 discusses the key design dimensions for continuous analyses. Section 3.3 presents the design of Codebase Analysis and Section 3.4 presents Solstice, which extends and instantiates Codebase Analysis design for Eclipse. Section 3.5 presents our experiments and results. Section 3.6 concludes with the contributions.

3.1 Definitions

In order to explain Codebase Analysis, we first define several concepts, including what it means for an analysis to be continuous.

A *snapshot* is the state of the source code of a software program at a point in time. An *analysis* is a computation on a snapshot that produces a result. An *offline analysis* is an analysis that requires no developer input. A *continuous analysis* is one that automatically computes an up-to-date result without the need for the developer to trigger it. Finally, a *pure analysis* is one that does not modify the snapshot on which it runs, while an *impure analysis* may. More formally:

Definition 1 (Snapshot)**.** A *snapshot* is a single developer's view of a program at a point in time, including the current contents of unsaved editor buffers. A unique snapshot is associated to each point in time.

Each of a developer's changes creates a new snapshot.

This chapter considers analyses that run on a single developer's codebase. Some analyses, such as conflict detection [21, 22], may need multiple developers' codebases for the same program.

Codebase Analysis is applicable to such analyses, although the definition of a snapshot would need to be extended.

Definition 2 (Analysis)**.** An *analysis* is a function $A\colon S \to R$ that maps a snapshot $s \in S$ to a result $r \in R$: $A(s) = r$.

Definition 3 (Offline analysis)**.** An *offline analysis* is an analysis that requires no human input during execution.

For example, a rename refactoring is not an offline analysis because each execution requires specifying a programming element (e.g., a variable) and a new name for this element. An offline analysis may require human input for one-time setup, such as setting configuration parameters or the location of a resource.

Definition 4 (Analysis implementation)**.** An offline *analysis implementation A_o* for an analysis A is a computer program that, on input snapshot s, produces $r = A(s)$.

We denote as $T_{A_o}(s)$ the time it takes an analysis implementation A_o to compute $r = A(s)$ on a snapshot s.

It is our goal to convert an offline analysis implementation A_o into a continuous analysis implementation A_c that executes A_o internally.

Definition 5 (ε-continuous analysis implementation)**.** Let A be an offline analysis, and let t_s be the time at which snapshot s comes into existence. An *analysis implementation A_c* that uses A_o is ε-*continuous* if $\exists \varepsilon_a, \varepsilon_s \leq \varepsilon$ such that for all snapshots s, both of the following are true:

1. A_c makes $r = A(s)$ available no later than $t_s + T_{A_o}(s) + \varepsilon_a$ if no new snapshot is created before this time. ε_a is the result delay: the time it takes to interrupt an ongoing analysis (A_o) execution, apply any pending edits to the copy codebase, restart the analysis, and deliver the results (e.g., to a UI or a downstream analysis). ε_a is independent of the underlying offline analysis run time since it does not include $T_{A_o}(s)$.
2. For all times after $t_{s+1} + \varepsilon_s$, A_c indicates that all results for s are stale. ε_s is the staleness delay: the time it takes to mark the displayed results as stale after the moment they become stale.

We often refer to ε-continuous analyses as simply *continuous*, implying that an appropriately small ε exists. For simplicity of presentation, our definition of a continuous analysis assumes the most eager policies for handling concurrent developer edits and displaying stale results. Sections 3.2.2 and 3.2.3 explore other policies.

To modify the developer's code simultaneously with development, the impure analysis must maintain a copy, which makes it particularly challenging to convert an *impure* offline analysis to a continuous analysis. Our approach handles both pure and impure analyses.

Definition 6 (Pure/impure analysis implementation)**.** An analysis implementation A_o is *pure* iff its computation on a snapshot s does not alter s. An *impure* analysis implementation may alter s.

Running a test suite is an example of a pure analysis because it does not alter the source code. Mutation analysis — applying a mutation operation to the source code and running tests on this mutant — is an impure analysis. An analysis that performs source code instrumentation is a special case of an impure analysis. An analysis that performs run-time instrumentation of a loaded binary is pure. Note that an impure analysis only alters the source code temporarily, while computing the results. Once the results are computed, the impure analysis or Codebase Analysis must revert the source code to its initial state.

3.2 Key Design Dimensions for a Continuous Analysis Tool

Our approach to implementing a continuous analysis involves executing an offline analysis internally. This section discusses three key design dimensions for continuous analysis tools that are converted from their offline analyses. Section 3.2.1 investigates what can trigger a continuous analysis tool to internally run the offline analysis, Section 3.2.2 investigates when a continuous analysis tool can abort an offline analysis execution, and Section 3.2.3 investigates how stale results are displayed to the developer. For each dimension, we discuss the advantages and disadvantages and provide examples from existing analysis implementations.

3.2.1 Triggering Analysis Execution

A continuous analysis may use four categories of triggers to start internal offline analysis executions: (1) whenever the snapshot changes in memory, (2) whenever the snapshot changes on disk, (3) periodically, and (4) other triggers. Additionally, analyses from each of the categories may be *delayed* and/or *overlapping*.

Memory-change triggers: The continuous analysis runs the internal offline analysis each time the program snapshot changes in memory, such as when the developer makes an edit in the IDE. A memory-change-triggered analysis provides feedback without requiring the developer to save the file. The Solstice Continuous Testing plug-in (described in Section 3.5.2) and Eclipse reconciler compiler[2] are examples of this category of analysis.

File-change triggers: The continuous analysis runs the internal offline analysis each time the program snapshot changes in the file system. A file-change-triggered analysis is motivated by the hypothesis that changes a developer saves to disk are more likely to be permanent than those merely made in memory. Finally, as file-system changes are less frequent than memory changes, these triggers can result in less resource use. However, waiting for changes to be saved to disk can delay computing pertinent analysis information. Eclipse provides a continuous analysis framework, called Incremental Project Builders [42]. Incremental Project Builders broadcasts the difference between two incremental builds on the file system, so that other analyses can access this difference and incrementally update results. Any continuous analysis Eclipse plug-in built using Incremental Project Builders, such as the FindBugs [55], Checkstyle [43], and Metrics [47] plug-ins, is a file-change-triggered continuous analysis.

Periodic and non-stop: The continuous analysis runs the offline analysis with a regular period. For example, the Crystal tool [22, 20] executes its analysis every 10 minutes. A variant of periodic analysis is a *non-stop* analysis, which runs every time the previous execution finishes. A periodic

[2]Eclipse contains two different compilers. The reconciler compiler operates on unsaved buffer content in order to give quick feedback; Eclipse calls it the “Java reconciler”. The incremental compiler operates on saved files and gives more complete and correct feedback about compiler errors whenever the user saves the document; Eclipse calls it the “incremental Java builder”.

analysis is most suitable when it is difficult to determine which actions may affect the analysis result.

Other triggers: Quality-control analyses often run before or after each version control commit. Pre-commit analyses prevent developers from committing code that breaks the build or test suite. These quality-control analyses must be fast since they would otherwise discourage the developer from making frequent commits. Building components and running unit tests that are directly affected by the changes are examples of such quality-control analyses. Post-commit analyses such as continuous integration notify developers soon after a bad commit. Continuous-integration analyses can be slower since they run separately from development, typically on a dedicated integration server. Building the software completely and running all integration tests are examples of continuous-integration analyses.

Analyses from each of the above categories may be *delayed* after the trigger before running the offline analysis. Eclipse's reconciler compiler is delayed until the developer pauses typing to avoid running the analysis during a burst of developer edits. The delay avoids executing the analysis on intermediate snapshots for which the results are less likely to be of interest to developers and are likely to become stale quickly: the delay thus also reduces analysis overhead. Delays are most appropriate for a memory-change-triggered analysis. Although file-change- and other-triggered analyses rarely use delays since actions such as saving a file or committing code already suggest good opportunities to run the offline analysis, these analyses might introduce delays for taking common developer patterns into consideration. For example, developers commit in bursts and a delay may avoid running the analysis on a snapshot that is about to be overridden by a new commit. Jenkins [82] supports a "quiet period" in which the builds are delayed after a commit to prevent an incomplete commit trigger a build failure.

Additionally, analyses from each of the above categories may be *overlapping*. Whenever a trigger fires while a previous offline analysis is still running, the continuous analysis has to decide whether to start a second, concurrent copy of the offline analysis. If the analyses are non-overlapping, the new offline analysis execution can be skipped, delayed until the current execution finishes, or started instead of finishing the previous execution (Section 3.2.2).

Surveying triggering in existing analyses: To determine how existing analysis tools, we surveyed Eclipse plug-ins on GitHub. We performed manual inspection of the documentation, automated analysis of the source code to find design patterns that indicate a continuous analysis, and manual inspection of the source code when necessary.

Of the 159 projects that match “Eclipse plug-in”, 47 implemented analysis tools. Of those 47, 21 (45%) were continuous: 1 (2%) was triggered by VCS commands, 12 (26%) were file-change-triggered and only 8 (17%) were memory-change-triggered. This suggests that despite the significant benefits of memory-change-triggered continuous analyses, they are difficult to implement, which motivates and justifies our work.

Of the 21 continuous analyses, 16 (8 file-change-triggered and 8 memory-change-triggered) extended Eclipse to handle languages other than Java, made possible in part by the relative simplicity of using the Incremental Project Builder [42], Xtext [152], and Reconciler [44] patterns. Similarly, we anticipate that our work will ease the creation and increase the number of memory-change-triggered continuous analysis tools for arbitrary analyses.

3.2.2 Abandoning the Ongoing Offline Analysis

When the developer edits the program while a continuous analysis is running the offline analysis, the continuous analysis can either (1) immediately interrupt the offline analysis execution without getting any results, potentially rerunning it on the latest snapshot, or (2) never interrupt the offline analysis, and finish running it on the snapshot, which is no longer the up-to-date development snapshot.

Immediately interrupt: The continuous analysis abandons the ongoing offline analysis immediately when a developer makes an edit. Such a continuous analysis is most suitable when the results of the analysis on an outdated snapshot have little or no value. An immediately-interrupting continuous analysis wastes no time executing on outdated snapshots. Quick Fix Scout [109] is an example of an immediately-interrupting continuous analysis.

Section 3.3.3 presents two designs for implementing immediately-interrupting continuous analyses by forcibly terminating the ongoing offline analysis.

Never interrupt: The continuous analysis continues to execute the offline analysis despite concurrent developer edits. To ignore the developer edits, the never-interrupting continuous analysis runs on a copy of a recent snapshot. An example is an analysis that runs after commits or nightly builds. A never-interrupting analysis is most suitable when the offline analysis takes a long time to run and when the results on a slightly outdated snapshot still has value to the developer. For example, a continuous integration server may complete running tests, while allowing developers to continue editing the program and make new commits. The test results are useful for localizing bugs. The alternative of interrupting the test suite execution every time the developer makes a change could mean the test suite rarely finishes and the developer rarely sees any analysis results.

Codebase Analysis enables never-interrupting continuous analyses by providing an isolated copy of the snapshot. More sophisticated interruption policies are possible. For example, when a conflicting developer edit takes place, an offline analysis could be permitted to complete its execution if that execution is estimated to be at least 50% done.

3.2.3 Handling Stale Results

A continuous analysis that does not interrupt its offline analysis may generate stale results. Furthermore, as the developer edits the code, displayed results may become stale.

We group continuous analyses by how they handle stale results into two categories: (1) immediately remove stale results, and (2) wait to remove stale results until new results are available. Analyses in either category can use cues to indicate that results are potentially stale and/or a new analysis is being run. It would also be possible to delay removing the analysis results or to create a separate (perhaps fast) analysis to check if stale results no longer apply, and remove them based on that analysis.

Immediately remove: The continuous analysis immediately removes the stale results and, potentially, indicates that the offline analysis is being rerun. This approach is appropriate if displaying stale results may lead to developer confusion. For example, showing a compilation error for code that the developer has already fixed may cause the developer to waste time re-examining the code. Examples include Quick Fix Scout [109] and most Eclipse analyses based on Incremental Project Builders, such as the Eclipse FindBugs and Checkstyle plug-ins. A never-interrupting, immediately-removing analysis may be a poor choice because if the developer edits the code while the analysis is running, the continuous analysis completes the offline analysis but never shows its results to the developer.

Display stale: The continuous analysis waits to remove the stale results until new results are available. Removing the old results may hinder a developer using them. For example, when fixing multiple compilation errors, the first keystroke makes all the results stale, but nonetheless the developer wants to see the error while fixing it and may want to move on to another error while the code is being recompiled. If the developer edits one part of the code, then all the analysis results technically become stale, but the developer may know that analysis about unaffected parts of the code remain correct. The Crystal tool [22, 20] is an example of a display-stale analysis, because it visually identifies results as potentially stale.

Unless otherwise noted, when we describe an ε-continuous analysis implementation, we mean a memory-change-triggered, non-overlapping, immediately-interrupting, immediately-removing analysis.

3.3 Codebase Analysis

This section describes our Codebase Analysis design, and how it addresses the challenges of isolation and currency.

3.3.1 Codebase Analysis Architecture

Codebase Analysis converts an offline analysis A_o into a continuous analysis A_c while addressing the two major challenges to creating continuous analysis tools: isolation and currency.

Isolation ensures that the developer's code changes and the execution of the offline analysis happen simultaneously without affecting each other. The developer should be isolated from A_c: A_c

should neither block the developer nor change the code as the developer is editing it (even though an impure A_o may need to change the code). Additionally, A_c should be isolated from the developer: developer edits should not alter the snapshot in the middle of an A_o execution, potentially affecting the results.

Despite isolation between the developer and A_c, *currency* requires A_c to react quickly to developer edits and to A_o results. Whenever the developer makes an edit, A_c should be notified so that it can mark old results as stale, terminate and restart A_o, or take other actions. A_c should react to fine-grained changes in the developer's editor's buffer, without waiting until the developer saves the changes to the file system nor commits them to a repository. A_c should also react quickly to A_o results, making them promptly but unobtrusively available to the developer or to downstream analyses.

Eclipse provides the Jobs API [48], which allows the UI thread to spawn an asynchronous task, execute it in the background, and eventually join back to the UI thread. Eclipse's incremental compiler and continuous analysis plug-ins implemented on Incremental Project Builders use the Jobs API. Codebase Analysis does not follow this approach because the Jobs API does not support the goals of isolation and currency. The Jobs API does not maintain a separate copy of the program, so any code changes by an impure analysis would interfere with the developer, thus failing to provide isolation. The Jobs API does not have direct access to an editor-buffer-level representation of the program, so to provide currency, the continuous analysis would have to combine active editor buffers with the file representation of the program.

Codebase Analysis addresses the isolation challenge by creating and maintaining an in-sync copy of the developer's codebase. Codebase Analysis addresses the currency challenge by providing notifications for events that occur in the developer's IDE and in A_c; these events can trigger terminating and restarting A_o and updating the UI.

Figure 3.1 shows the architecture of Codebase Analysis. The IDE API generates events for all developer actions, including changes to the code. Codebase Analysis keeps a queue of these events, and applies them to the copy codebase. Meanwhile, A_c can pause the queue, run A_o on the copy snapshot, collect A_o results, resume the queue, and update the results shown to the developer.

Codebase Analysis also notifies A_c about new developer actions, so that A_c may decide to interrupt, alter, or continue A_o's execution.

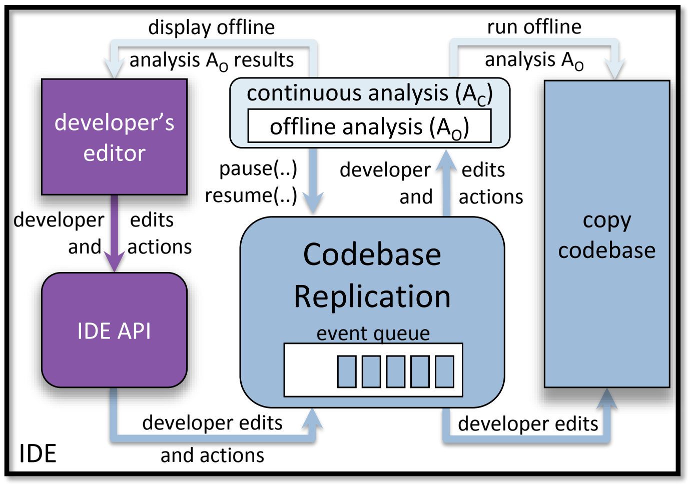


Figure 3.1: Codebase Analysis architecture. Codebase Analysis (blue) facilitates communication between A_c (light blue) and a developer's IDE (dark purple) via asynchronous events.

Codebase Analysis supports multiple A_c using the same copy codebase, via the readers-writers lock protocol. Codebase Analysis runs one impure A_c or multiple pure A_c in parallel. For example, a developer might run continuous testing and FindBugs in parallel to obtain both dynamic information — test results — and static information — FindBugs warnings. Running multiple pure A_c in parallel amortizes the already-low overhead (see Section 3.5.1). Codebase Analysis guarantees that, even with multiple A_c, at any given time, the copy codebase can be modified by at most one analysis, and the copy codebase never changes during a pure analysis execution.

For exposition purposes, this dissertation introduces the Codebase Analysis design with one copy codebase. A Codebase Analysis implementation can maintain multiple copy codebases and run multiple impure A_c in parallel by running each impure A_c on a separate copy codebase.

3.3.2 Ensuring Isolation and Currency

In addition to Codebase Replication's principles, Codebase Analysis employs two more principles to overcome the challenges of isolation and currency: exclusive ownership, and staleness detection.

Exclusive ownership: Changing the snapshot in the middle of an execution may cause A_o to produce incorrect results or to crash. The situation also arises if multiple A_o run on the same code at once, and at least one of them is impure. Codebase Analysis allows one impure A_c at a time to claim exclusive access to a copy snapshot while its A_o executes, excluding all other analyses and pausing synchronization updates with the developer's buffer.

Staleness detection: A_o results from an old snapshot might become stale as a result of the developer's changes. If a change occurs while A_o executes, the result may already become stale by the time A_o completes. When a change occurs, Codebase Analysis notifies A_c and allows it to choose to finish executing or to terminate A_o.

Section 3.5 will revisit how well our design and implementation satisfy these requirements.

3.3.3 Improving Currency with Analysis Termination

To improve analysis input currency, when there is an edit that conflicts with an ongoing analysis execution, Codebase Analysis has the ability to terminate the ongoing analysis execution, apply the conflicting edits to the copy codebase, and rerun the analysis on the updated codebase. Proper termination of an ongoing analysis requires additional support from either A_o or A_c. This section discusses two designs: one in which the support is provided by A_o and one in which the support is provided by A_c.

$\mathbf{A_o}$-provided termination support: The first way to provide external support for termination is to require external interruption support from the offline analysis A_o. When Codebase Analysis interrupts an ongoing analysis execution, A_o is expected to abandon its execution and do any cleanup that is needed in a timely manner, such as reverting modifications to the source code, databases, file pointers, and class loaders. If A_o provides external interruption support, A_c requires no changes. It just interrupts A_o when a conflicting developer edit is detected.

$\mathbf{A_c}$-provided termination support: The second way to provide external support for termination is to design A_c to never interrupt the offline analysis A_o, instead executing A_o multiple times on different chunks of the codebase (e.g., one execution per file). A_c then needs to compose the individual results into a single analysis result for the whole program. Each time A_o finishes executing on a chunk, A_c checks for interrupts. If there is an interrupt, A_c abandons computing the result for the whole program, cleans up, and returns ownership to Codebase Analysis. This approach does not require any modifications to A_o. However, it is only applicable when A_o is modular (can be split up to work on program chunks) and executing it on individual chunks is much faster than on the whole program.

To enable A_c-provided termination support, Codebase Analysis supports a *step-based execution model* that handles interruption and termination, making it easy to write an interruptible A_c for a modular A_o. The continuous analysis A_c creates a sequence of *steps*, each one an atomic unit of work that executes in a reasonable amount of time, such as several seconds. There are two kinds of steps: (1) RUN steps that represent normal analysis execution, and (2) CLEANUP steps that clean up side effects. Codebase Analysis maintains a worklist of steps, executes them in order, and checks for interruptions after each step execution. When it detects an interruption, such as a developer edit, Codebase Analysis ignores the remaining RUN steps and executes the remaining CLEANUP steps.

Consider a continuous testing analysis tool (Section 3.5.2) that runs all tests in the project and displays the results to the developer. The tool would not be responsive to the developer's changes if it finishes executing the entire test suite before checking for new developer changes. The step-based execution model improves the tool's responsiveness by checking for changes more often, for example, after each class's tests finish.

For a modular offline analysis, executing it on a chunk of code corresponds to adding one RUN step to the worklist. For example, continuous testing adds the following steps to the worklist: one RUN step that creates a new class loader and identifies the test classes that JUnit can run (concrete classes with at least one test), one RUN step per test class that runs JUnit on that class, and one CLEANUP step, which releases the resources used by the new class loader. For the above-mentioned continuous testing implementation, Codebase Analysis checks for conflicting edits after each step. If there is an interruption, Codebase Analysis ignores remaining RUN steps, but executes the CLEANUP step, which ensures that the new class loader does not leak memory.

Assuming that executing each step is bounded by τ time, the step-based analysis execution approach guarantees that the offline analysis execution can be terminated safely in $(c+1)\cdot\tau$ time, where c is the number of CLEANUP steps added to the worklist before the analysis is interrupted. We anticipate that for most analyses, c will be small.

Our Eclipse-based Codebase Analysis prototype Solstice, which we describe next in Section 3.4, supports the step-based execution model, and all the Solstice-based plug-in analyses from Section 3.5.2 use it. Support for the step-based execution model required, on average, only an extra 100 LoC.

3.4 Solstice: Codebase Analysis for the Eclipse IDE

To evaluate Codebase Analysis, we built Solstice, an Eclipse-based, open-source Codebase Analysis prototype. Solstice is available at `https://bitbucket.org/kivancmuslu/solstice/` publicly. This section describes Solstice (Section 3.4.1), explains how to implement continuous analysis tools using Solstice (Section 3.4.2), and describes one such implementation (Section 3.4.3). Later, Section 3.5.2 describes our experience using Solstice to develop four continuous analysis tools.

To the best of our knowledge; Solstice is the first framework that aids implementing memory-change-triggered analysis tools for arbitrary source and binary code analyses, which would otherwise be considerably more difficult to build.

3.4.1 Solstice Implementation

This section explains the Solstice implementation and refines Codebase Analysis with Eclipse-specific concerns.

Figure 3.2 illustrates Solstice's architecture. Solstice consists of two parts. `Solstice server` runs on the developer's Eclipse and is responsible for listening to the developer's actions. `Solstice client` runs on a background Eclipse (which we describe next) and is responsible for keeping the copy codebase in sync and managing the ownership of the copy codebase. Solstice-based continuous analysis tools use `Solstice client` for their computation logic and `Solstice server` for their visualization logic and to interact with the developer.

The Eclipse API allows each Eclipse process to be associated with (and have access to) only one workspace. Solstice interacts with two Eclipse processes running at once: the developer's normal Eclipse, which manages the developer's workspace, and a second, background Eclipse,

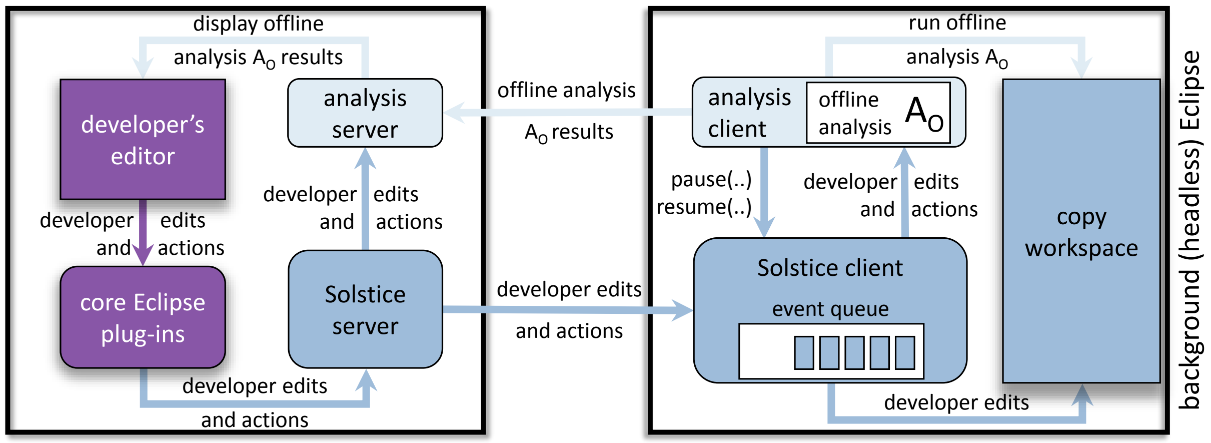


Figure 3.2: Solstice architecture as an instantiation of the Codebase Analysis architecture (Figure 3.1) for Eclipse. Solstice observes the workspace in the developer's Eclipse and creates a new Eclipse process to manage the copy workspace. Solstice and the continuous analysis (A_c) each consist of two components, a server (`Solstice server` for Solstice and `analysis server` for A_c) that interacts with the developer's Eclipse, and a client (`Solstice client` for Solstice and `analysis client` for A_c) that interacts with the copy Eclipse. The developer's Eclipse (dark purple) generates events for developer actions, including edits. `Solstice server` (blue) sends these events to `Solstice client` (blue) and notifies A_c (light blue) of the actions. `Solstice client` stores these actions temporarily in the event queue, applies the edits to the copy workspace, notifies A_c of these actions, and provides a pause-resume API for managing exclusive ownership of the copy workspace. A_c (light blue) interacts with the developer's editor, Solstice, and the copy Eclipse. `Analysis client` runs A_o on the copy workspace and sends the results to `analysis server`. `Analysis server` modifies the developer's editor accordingly and implements staleness logic.

which runs `Solstice client` and maintains the copy workspace (and with it, the copy codebase). The background Eclipse is headless — it has no UI elements and the developer never sees it. The copy workspace resides in a hidden folder on disk. The Solstice implementation maintains one copy codebase. It runs all pure A_c in parallel, and each impure A_c in isolation.

Each time the developer starts Eclipse, Solstice executes an *initialization synchronization protocol* that briefly blocks the developer and ensures that the copy workspace is in sync with the developer's workspace. The first time the developer uses Solstice, the initialization synchronization protocol acts as a full synchronization and creates a complete copy of the developer's workspace on disk. Future executions verify the integrity of the files in the copy workspace through checksum and update the files that were added, removed, or changed in the developer's workspace outside of the IDE.

After the initialization synchronization protocol, `Solstice server` attaches listeners to the developer's Eclipse. The listeners track edits to the source code, changes to the current cursor location, changes to the currently selected file, changes to the currently selected Eclipse project, invocations of Quick Fix, proposals offered for a Quick Fix invocation, and selections, completions, and cancellations of Quick Fix proposals. Developer actions that alter the code generate both a developer action event (e.g., to say that the developer clicked on a menu item) and an edit event that encodes the code changes. `Solstice server` sends the developer's events to the `Solstice client`, which makes the incoming events available to the continuous analysis tool through the observer pattern and applies all edits on the developer's workspace to the copy workspace.

3.4.2 Building Solstice-Based Tools

This section explains how to use Solstice to build a continuous analysis tool A_c based on an offline analysis implementation A_o. We refer to the *author* as the person developing A_c, and to the *developer* as the person later using A_c.

To implement A_c, the author specifies: (1) how A_c computes the results, (2) how A_c interacts with the developer, (3) the information that needs to be communicated between the server and the client components of A_c, and (4) how A_c handles stale results.

(1) The author specifies A_c computation logic — how A_o runs and produces results. The computation logic is implemented as an Eclipse plug-in that interacts with `Solstice client`, represented as `analysis client` in Figure 3.2. The computation logic always runs on the background Eclipse using the contents of the copy workspace.

Most analyses must verify some pre-conditions before running on a codebase. Solstice API contains analysis steps that simplify this verification process for common pre-conditions. For example, the author can use `ProjectCompilesStep` to ensure that the codebase has no compilation errors or `ResourceExistsStep` to ensure that a particular resource (e.g., test folder) exists. If a step's pre-conditions fails, Solstice abandons the analysis and shows a descriptive warning message to the developer.

As an additional contingency mechanism for infinite loops due to bugs in the analysis or dynamic execution of unknown code, Solstice lets the author specify a *timeout* for each step. If a step takes longer than its timeout, Solstice assumes that the analysis went into an infinite loop and terminates the analysis, including the remaining steps.

(2) The author specifies the interaction logic — how A_c shows results and interacts with the developer. The interaction logic is implemented as an Eclipse plug-in that interacts with `Solstice server`, represented as `analysis server` in Figure 3.2. The interaction logic runs on the developer's Eclipse using the developer's editor.

The same way Eclipse manages the life-cycle of its plug-ins, Solstice manages the life-cycle of A_c: each A_c starts after Solstice starts (when the developer opens Eclipse) and terminates before Solstice terminates (when the developer closes Eclipse). The author does not need to create and manage a thread for A_c, as Solstice takes care of these details.

For the rest of the section, we assume that A_c interacts with the developer. Continuous analysis tools that do not interact with the developer (e.g., an observational A_c that only logs developer actions) do not need an `analysis client` component: `Solstice client` duplicates all developer edits and A_c (`analysis client`) can access those events directly from `Solstice client` via listeners.

(3) The author specifies the communication between `analysis client` and `analysis server`. The analysis results generated by `analysis client` need to be sent to `analysis server` to be displayed to the developer, as shown in Figure 3.2. The communication does not have to be one-directional (although the example communication shown in Figure 3.2 is). For example, the `analysis server` can allow the developer to modify A_c settings, which it would then send to the `analysis client`.

The Solstice API trivializes the inter-process communication between the `analysis client` and the `analysis server`. The author can invoke `sendMessage(...)` to communicate a `Serializable` Java object from the `analysis client` to the `analysis server` or vice versa. Solstice takes care of all low-level networking details, deserializes the object on the receiver, and executes a method that processes the object.

(4) The author writes the logic for handling potentially-stale A_o results. Solstice timestamps

Program	Workspace Size (MB)	Code Size (KLoC)	Sync Time (seconds)			
			Empty Copy		In-sync Copy	
			μ	σ	μ	σ
CrosswordSage [35]	17	4	0.3	0.2	0.1	0.1
ASMX [5]	26	43	1.8	0.4	0.5	0.1
Voldemort [148]	55	160	6.4	0.8	1.3	0.2
JDT [46]	164	1,694	124.0	10.8	8.6	1.1

Figure 3.3: Solstice initial synchronization protocol performance. Each cell is the mean of 20 executions.

every developer action and edit, A_o start, and A_o finish, to ensure that no event is lost and that Solstice knows to which snapshot an A_o result applies. Solstice supports all the policies discussed for abandoning A_o (Section 3.2.2) and handling stale results (Section 3.2.3). Solstice provides APIs for common scenarios, such as removing A_o results with each developer edit. To specify the staleness behavior of A_c, the author needs to set the value of one `boolean` argument. Solstice takes care of all low-level details, such as attaching multiple listeners to Eclipse to detect resource changes and updating the analysis visualization while handling potentially stale results.

3.4.3 An Example Solstice Continuous Analysis Plug-in

Suppose an author wants to use Solstice to build an A_c using an A_o. The author decides that A_c will be never-interrupting (Section 3.2.2) and display-stale (Section 3.2.3): when the developer makes a change while A_o executes, A_o might as well finish, and A_c will display potentially stale results to the developer, with an indicator.

The author would have to write the following interaction logic for A_c (`analysis server`):

```
class Server extends AnalysisServer {
  public Server() {
    super(true /*Display potentially stale results with an indicator*/);
  }
  // Implement one of the following two methods:
  /** Convert results to human-readable text that will be displayed
    * in the analysis view. */
  String resultToText(Result result) {...}
  /** Convert results to Eclipse markers that will be displayed
    * in the analysis marker view and in the source code. */
  IMarker[] resultToMarkers(Result result) {...}
}
```

`Server` passes `true` to the `AnalysisServer` constructor, which makes Solstice display potentially stale results with a special indicator. Solstice calls `resultToText(...)` and `resultToMarkers(...)` with A_o results (`Result`) received from the `analysis client`. The author needs to implement at least one of these methods to transform `Result` into a human-readable text or Eclipse markers, which Solstice uses to automatically update the contents of the corresponding Eclipse view.

The author would also have to write the following computation logic for A_c (`analysis client`):

```
class Analysis extends ResourceBasedAnalysis {
  List<Step> getSteps() {
    List<Step> steps = new ArrayList<>();
    steps.add(new RunStep() {
      void run() {
        Result result = runOfflineAnalysis();
        generateResult(result);
    }});
    return steps;
  }
}
```

`Analysis` extends `ResourceBasedAnalysis`, which makes Solstice rerun A_o each time Solstice applies all developer edits to the copy workspace and the copy workspace is up to date. Under the step-based semantics, Solstice executes the analysis steps returned from `getSteps()`. Each step can invoke `generateResult(Result)`, which makes Solstice automatically send this `Result` to the `analysis server`.

3.5 Evaluation

To evaluate Solstice, we empirically measured its performance overhead (Section 3.5.1), determined the ease of using Solstice by implementing four proof-of-concept continuous analysis tools (Section 3.5.2), observed developers' interaction with continuous analysis tools in two case studies (Section 3.5.3), and compared Solstice to other methods of implementing IDE-integrated continuous analyses (Section 3.5.4).

Operation Name	Operation Size	Initial File Size (chars)	IDE Over-head (s)	Sync Delay (s)	Operation Name	Operation Size	Initial File Size (chars)	IDE Over-head (s)	Sync Delay (s)
Text Insert	1	1	0.001	0.002	Text Delete	1	1	0.001	0.002
		100	0.001	0.002			100	0.001	0.002
		1,000	0.001	0.002			1,000	0.001	0.002
		10,000	0.002	0.002			10,000	0.003	0.002
	100	1	0.001	0.002		100	1	0.001	0.002
		100	0.001	0.002			100	0.001	0.002
		1,000	0.001	0.002			1,000	0.001	0.002
		10,000	0.002	0.003			10,000	0.002	0.002
Text Edit Summary								$\leq$ 0.003	$\leq$ 0.003
File Add	1	1,000	0.001	0.001	File Remove	1	1,000	0.001	0.001
	100		0.102	0.157		100		0.056	0.106
	1,000		1.464	1.305		1,000		0.566	2.491
File Edit Summary								grows linearly with size	

Figure 3.4: The Solstice-induced overhead on developer edits for keeping the copy workspace in sync. Text operations of size 1 are single keystrokes, and larger text operations add, replace, or remove 100 consecutive characters at once to represent cut, paste, and tool applications, such as applying a refactoring or an auto-complete. File operations of size 1 are manual file generation, copy, and removal, and larger file operations represent copying, removing, or importing a directory or an entire Eclipse project. "IDE Overhead" measures the overhead imposed on the responsiveness of the IDE, and "Sync Delay" measures the delay before the copy workspace is up to date. For each text operation experiment, we executed the operation 100 times and took the average to reduce external bias, such as JVM warmup.

3.5.1 Solstice Performance Evaluation

An effective continuous analysis should meet the following requirements:

Low initialization overhead: The developer should not be blocked too long during startup (Section 3.5.1).

Low synchronization overhead: While using the IDE, the developer should experience negligible overhead (Section 3.5.1).

High analysis input currency: The delay after an edit before an analysis can access an up-to-date program in the copy codebase should be small (Section 3.5.1).

This section presents the results of performance experiments addressing these three requirements. The experiments were executed on a MacBook Pro laptop (Mac OS X 10.9, i7 2.3 GHz quad core, 16 GB RAM, SSD hard drive). Solstice ran with a 512 MB RAM limitation for each of the server and client components.

Initial Synchronization Protocol Cost

Every time the developer runs Eclipse, Solstice executes a blocking initial synchronization protocol (recall Section 3.4) to ensure that the copy workspace is in sync with the developer's workspace.

This is required because Solstice does not track changes to the developer's workspace when Eclipse is not running.

We have tested Solstice's initial synchronization protocol using four different workspace contents (Figure 3.3). For each setting, we created a workspace with one program and invoked the initial synchronization protocol for two extreme cases: full synchronization and no synchronization. In the full case, the copy workspace is empty, which requires Solstice to copy the entire workspace. In the none case, the copy workspace is already in sync, which requires Solstice to only verify that the copies are in sync using checksums. Since developers make most of their code changes within an IDE, we expect most invocations of Solstice after the first one to resemble the none case.

Figure 3.3 shows that Solstice has low synchronization overhead. This could be further reduced by a lazy initial synchronization protocol that only processes the active Eclipse project and its dependencies, not all projects in the program (for example, JDT consists of 29 Eclipse projects from eclipse.jdt, eclipse.jdt.core, eclipse.jdt.debug, and eclipe.jdt.ui).

Solstice would have been easier to implement if it always built a brand new copy of the workspace on Eclipse startup. There would be two main sources of overhead:

1. Copying the files. For the JDT workspace, containing 16,408 files, this takes 30 seconds ($\sigma = 1.1$ sec.).
2. Creating an Eclipse project and importing it into the workspace. Eclipse creates metadata for the project and indexes project files. For the JDT workspace, this takes 74 seconds ($\sigma = 2.5$ sec.).

This is 12 times slower than Solstice's incremental synchronization, which takes only 8.6 seconds for the JDT workspace.

IDE Synchronization Overhead

Solstice tracks all developer changes at the editor buffer level. The "IDE overhead" column of Figure 3.4 shows, for the most common developer actions, the IDE overhead that Solstice introduces when the action is initiated programmatically. The overhead is independent of the edit size and is no more than 2.5 milliseconds.

Even adding or removing 1000 files incurs modest overhead that is similar to the 1.085 seconds that Eclipse takes to import similar-sized project (org.eclipse.jdt.core, 1205 files).

Analysis	Lines of Code					Dev.	Evaluation Subject Program				A_o Run	ε_a	ε_i
	UI	IPC	Core	Other	Total	Time (h)	Name	Version	KLoC	KNCSL	Time (s)	(s)	(s)
FindBugs	240	18	239	22	519	25	Voldemort	1.6.4	160	115	74	0.033	0.002
PMD	265	18	224	22	529	4					25	0.102	0.004
Check Sync.	160	39	324	22	545	24	Java Grande	1.0	10.5	4.8	293	0.073	0.004
Testing	581	184	458	22	1245	25	Commons CLI	1.3	10.5	5.8	0.01	0.108	0.003

Figure 3.5: Summary of four Solstice-based continuous analysis tools. Each tool consists of "UI" code for basic configuration and result visualization, "IPC" code for serialization, "Core" code for setting up and running the offline analysis A_o, and "Other" code for extension points. Code sizes are larger than reported in [105] because of new UI functionality and support for the step-based execution model. The continuous PMD analysis took little development time due to similarities to FindBugs, which was developed before it. Each tool is ε-continuous and the table reports the maximum observed ε values. The experiments used PMD's `java-basic` ruleset and all of Commons CLI's 361 tests. The Check Synchronization evaluation considers 11 (out of 31) of the Java Grande benchmark programs (main classes). 14 programs could not be executed by Check Synchronization and 6 programs took longer than 5 minutes and were excluded due to time considerations.

Copy Codebase Synchronization Delay

To allow A_c to access the up-to-date version of the developer's code, Solstice must quickly synchronize the copy workspace. The "Sync delay" column of Figure 3.4 shows the delay Solstice incurs during synchronization for the most common developer operations. Synchronizing text edits takes no more than 2.5 milliseconds. Thus, Solstice provides A_c access to the developer's code that is no more than 2.5 milliseconds old. Importing and deleting a 1000-file project takes longer, up to 2.5 seconds, but since these operations are rare and already take several seconds for Eclipse to execute, the Solstice delay should be acceptable.

Summary

Our performance analysis demonstrates that Solstice introduces negligible overhead to the IDE, does not interrupt the development process (except during startup) and provides access to an up-to-date copy codebase with negligible delay.

3.5.2 Solstice Usability Evaluation

We have used Solstice to build four continuous analysis Eclipse plug-ins, each using an existing offline analysis implementation. This section describes these implementations and reports on the building experience.

Figure 3.5 summarizes the continuous analysis tools built on Solstice. The epsilon values in Figure 3.5 are computed by instrumenting Solstice to timestamp the moments when:

t_e: the developer makes an edit,
t_{as}: Solstice starts running A_o,
t_{ae}: Solstice finishes running A_o,
t_d: Solstice displays analysis results to the developer,
$t_{e'}$: the developer makes a new edit,
t_s: Solstice marks analysis results as stale (after the new edit)

After an analysis computed its initial results, we made 10 small edits in Eclipse (ranging a few lines to adding/remove one method) that produce different analysis results. For each analysis execution, we computed $\varepsilon_i = t_s - t_{e'}$ and $\varepsilon_a = (t_{as} - t_e) + (t_d - t_{ae})$ and Figure 3.5 displays their maximum.

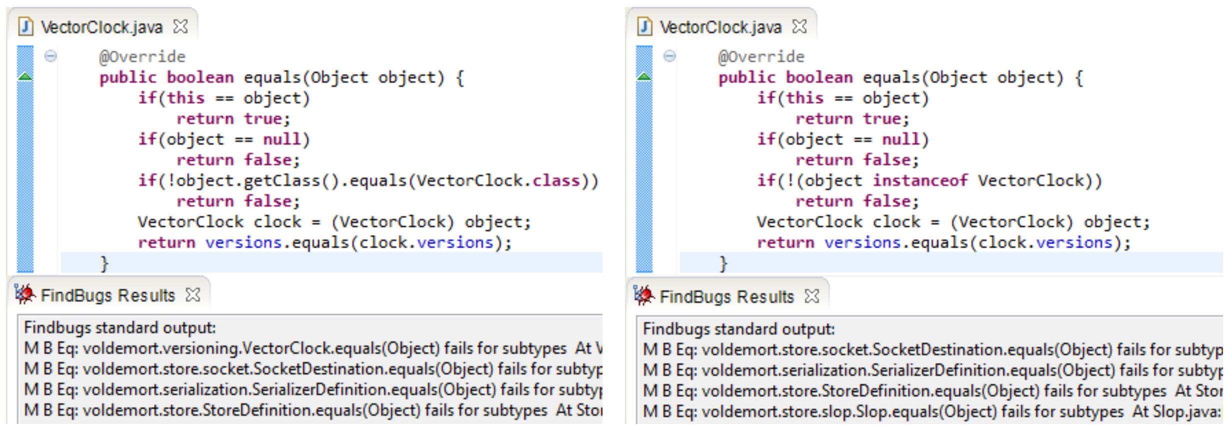
This section presents each continuous analysis as a separate Eclipse plug-in. Solstice supports running multiple pure A_c in parallel.

Continuous FindBugs

FindBugs is a static analysis tool that finds common developer mistakes and bad practices in Java code, such as incorrect bitwise operator handling and incorrect casts. FindBugs has found bugs in open-source software, is useful to developers, and is extensible with new defect patterns [76]. It is available as a command-line and a GUI tool, an Ant task extension, and an Eclipse plug-in [55].

The FindBugs Ant task extension and Eclipse plug-in can automate FindBugs invocations, but both fall short of being ε-continuous according to Definition 5 in Section 3.1. The Ant task extension executes only with each Ant build. The Eclipse plug-in has two FindBugs implementations. The developer has to manually invoke the complete FindBugs that analyzes the whole project. There is also a lighter Eclipse-Incremental-Project-Builders version that is disabled by default. This lightweight version automatically recomputes the FindBugs warning for the current editor file whenever the developer saves outstanding changes on the editor file. Both tools require the developer to perform an action to run, and neither reacts to changes made to the editor buffer. Further, since changes to one file may affect the analysis results of another, the lightweight mode of FindBugs plug-in may miss warnings.

We have used Solstice to build a proof-of-concept, open-source continuous FindBugs plug-in, available at `https://bitbucket.org/kivancmuslu/solstice-continuous-findbugs/`. The plug-in uses the command-line FindBugs to analyze the `.class` files for all the classes in the currently active Eclipse project and all their dependent libraries. The plug-in's simple visualization displays the FindBugs warnings in an Eclipse view [49], which is a configurable window similar



VectorClock.java

```
    @Override
    public boolean equals(Object object) {
        if(this == object)
            return true;
        if(object == null)
            return false;
        if(!object.getClass().equals(VectorClock.class))
            return false;
        VectorClock clock = (VectorClock) object;
        return versions.equals(clock.versions);
    }
```

FindBugs Results

```
Findbugs standard output:
M B Eq: voldemort.versioning.VectorClock.equals(Object) fails for subtypes  At V
M B Eq: voldemort.store.socket.SocketDestination.equals(Object) fails for subtyp
M B Eq: voldemort.serialization.SerializerDefinition.equals(Object) fails for subty
M B Eq: voldemort.store.StoreDefinition.equals(Object) fails for subtypes  At Sto
```

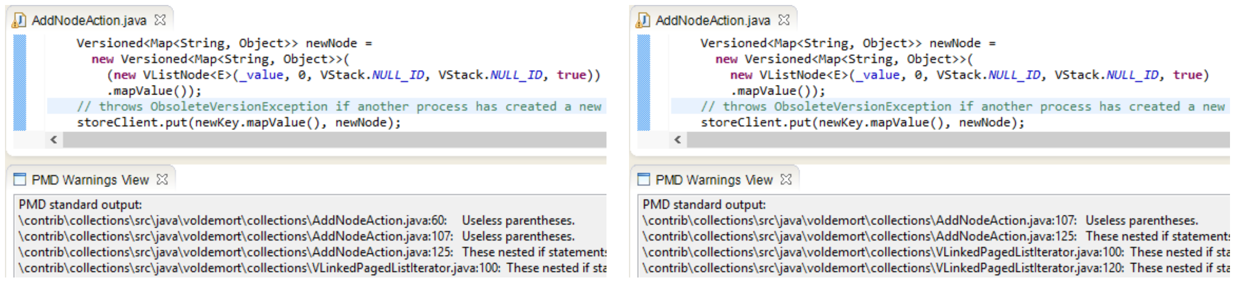
VectorClock.java

```
    @Override
    public boolean equals(Object object) {
        if(this == object)
            return true;
        if(object == null)
            return false;
        if(!(object instanceof VectorClock))
            return false;
        VectorClock clock = (VectorClock) object;
        return versions.equals(clock.versions);
    }
```

FindBugs Results

```
Findbugs standard output:
M B Eq: voldemort.store.socket.SocketDestination.equals(Object) fails for subtyp
M B Eq: voldemort.serialization.SerializerDefinition.equals(Object) fails for subtyp
M B Eq: voldemort.store.StoreDefinition.equals(Object) fails for subtypes  At Stor
M B Eq: voldemort.store.slop.Slop.equals(Object) fails for subtypes  At Slop.java:
```

Figure 3.6: Continuous FindBugs running on Voldemort. Both images show the top four warnings. The left screenshot shows the original Voldemort implementation; its first FindBugs warning suggests that the first use of `.equals(...)` is too restrictive. The developer changes `.equals(...)` to `instanceof` (right screenshot) and the top warning disappears without the developer saving the file or invoking FindBugs.



AddNodeAction.java

```
        Versioned<Map<String, Object>> newNode =
          new Versioned<Map<String, Object>>(
            (new VListNode<E>(_value, 0, VStack.NULL_ID, VStack.NULL_ID, true))
            .mapValue());
        // throws ObsoleteVersionException if another process has created a new
        storeClient.put(newKey.mapValue(), newNode);
```

PMD Warnings View

```
PMD standard output:
\contrib\collections\src\java\voldemort\collections\AddNodeAction.java:60:   Useless parentheses.
\contrib\collections\src\java\voldemort\collections\AddNodeAction.java:107:  Useless parentheses.
\contrib\collections\src\java\voldemort\collections\AddNodeAction.java:125:  These nested if statement
\contrib\collections\src\java\voldemort\collections\VLinkedPagedListIterator.java:100:  These nested if sta
```

AddNodeAction.java

```
        Versioned<Map<String, Object>> newNode =
          new Versioned<Map<String, Object>>(
            new VListNode<E>(_value, 0, VStack.NULL_ID, VStack.NULL_ID, true)
            .mapValue());
        // throws ObsoleteVersionException if another process has created a new
        storeClient.put(newKey.mapValue(), newNode);
```

PMD Warnings View

```
PMD standard output:
\contrib\collections\src\java\voldemort\collections\AddNodeAction.java:107:  Useless parentheses.
\contrib\collections\src\java\voldemort\collections\AddNodeAction.java:125:  These nested if statement
\contrib\collections\src\java\voldemort\collections\VLinkedPagedListIterator.java:100:  These nested if sta
\contrib\collections\src\java\voldemort\collections\VLinkedPagedListIterator.java:120:  These nested if sta
```

Figure 3.7: Continuous PMD running on Voldemort. Both images show the top four warnings. The left screenshot shows the original Voldemort implementation; its first PMD warning suggests that the parentheses around `new` are unnecessary. The developer removes these parentheses (right screenshot) and the first warning disappears, without the developer saving the file or invoking PMD.

to the Eclipse Console. The plug-in immediately removes potentially stale warnings and recomputes warnings for the up-to-date codebase. Figure 3.6 shows two continuous FindBugs plug-in screenshots.

Continuous PMD

PMD [123] is a static Java source code analysis that finds code smells and bad coding practices, such as unused variables and empty catch blocks. It is available for download as a standalone executable and as plug-ins for several IDEs, including Eclipse. Like FindBugs, it is popular and well-maintained. Unlike FindBugs, PMD works on source code. The existing Eclipse plug-in is not continuous; the developer must right-click on a project and run PMD manually.

We have used Solstice to build a proof-of-concept, open-source continuous PMD plug-in, available at `https://bitbucket.org/kivancmuslu/solstice-continuous-pmd/`. The plug-in uses the command-line PMD to analyze the `.java` files for the currently active Eclipse project. The plug-in's visualization displays the PMD results in an Eclipse view. The plug-in immediately removes potentially stale results and recomputes results for the up-to-date codebase. Figure 3.7 shows two continuous PMD plug-in screenshots.

Continuous Check Synchronization

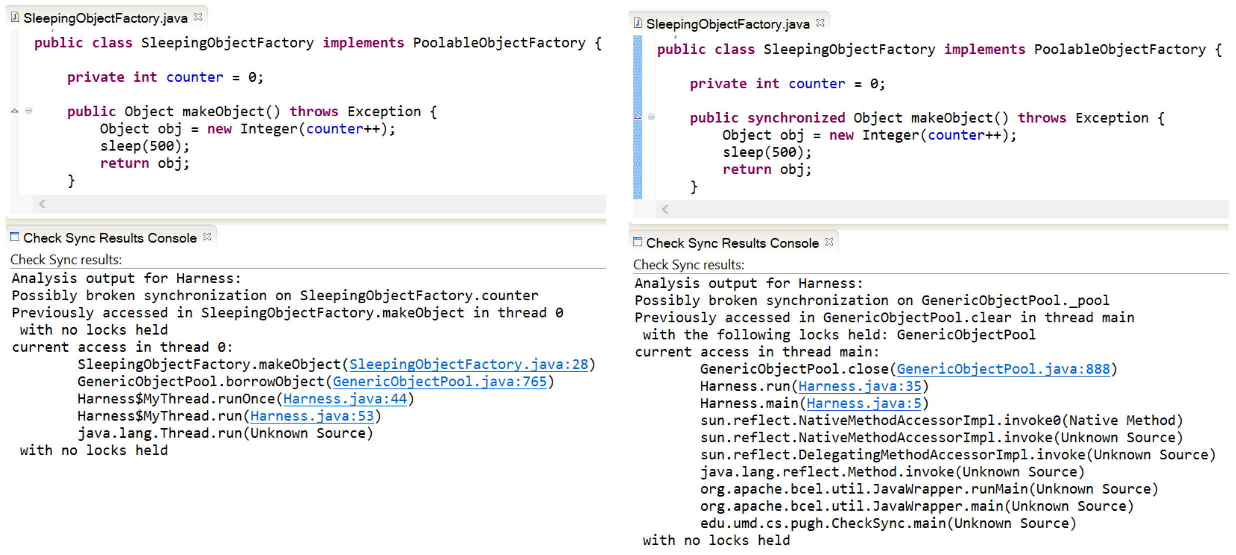
Check Synchronization [29], based on technology first introduced in Eraser [131], is a race detection tool that detects potentially incorrect synchronization using dynamic checks. The tool has not yet been integrated into any IDEs.

We have used Solstice to build a proof-of-concept, open-source continuous Check Synchronization plug-in, available at `https://bitbucket.org/kivancmuslu/solstice-continuous-check-synchronization/`. The plug-in searches for all classes with main methods inside the current project and runs the Check Synchronization tool on these classes. For each class with a main method, the plug-in shows the results to the developer through an Eclipse view. The plug-in immediately removes potentially stale results and recomputes new results for up-to-date codebase. Figure 3.8 shows two continuous Check Synchronization plug-in screenshots.

Continuous Testing

Continuous testing [130] uses otherwise idle CPU cycles to run tests to let the developer know as soon as possible when a change breaks a test. Continuous testing can reduce development time by up to 15% [128]. There are Eclipse [130, 78], Visual Studio [34], and Emacs [129] plug-ins for continuous testing. The original Eclipse plug-in [130] is ε-continuous, however it modifies Eclipse core plug-ins, making it difficult to update the implementation for new Eclipse releases; in fact, the plug-in does not support recent versions of Eclipse. By contrast, Solstice requires no modifications to the Eclipse core plug-ins and would apply across many Eclipse versions.

We have used Solstice to build a proof-of-concept, open-source continuous testing plug-in, available at `https://bitbucket.org/kivancmuslu/solstice-continuous-testing/`. The plug-in runs the tests of the currently active Eclipse project. The plug-in immediately removes potentially stale test results and recomputes the test results for up-to-date codebase. The plug-in's simple visualization displays the test results in an Eclipse view. Figure 3.9 shows two continuous testing plug-in screenshots.


SleepingObjectFactory.java

```
public class SleepingObjectFactory implements PoolableObjectFactory {

    private int counter = 0;

    public Object makeObject() throws Exception {
        Object obj = new Integer(counter++);
        sleep(500);
        return obj;
    }
```

Check Sync Results Console

Check Sync results:

```
Analysis output for Harness:
Possibly broken synchronization on SleepingObjectFactory.counter
Previously accessed in SleepingObjectFactory.makeObject in thread 0
 with no locks held
current access in thread 0:
        SleepingObjectFactory.makeObject(SleepingObjectFactory.java:28)
        GenericObjectPool.borrowObject(GenericObjectPool.java:765)
        Harness$MyThread.runOnce(Harness.java:44)
        Harness$MyThread.run(Harness.java:53)
        java.lang.Thread.run(Unknown Source)
 with no locks held
```

SleepingObjectFactory.java

```
public class SleepingObjectFactory implements PoolableObjectFactory {

    private int counter = 0;

    public synchronized Object makeObject() throws Exception {
        Object obj = new Integer(counter++);
        sleep(500);
        return obj;
    }
```

Check Sync Results Console

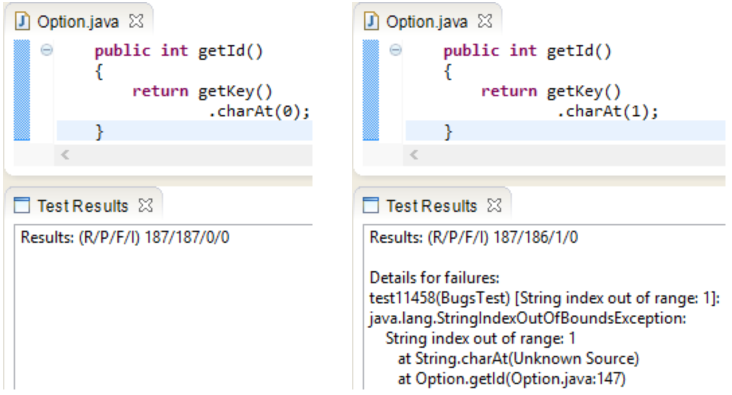
Check Sync results:

```
Analysis output for Harness:
Possibly broken synchronization on GenericObjectPool._pool
Previously accessed in GenericObjectPool.clear in thread main
 with the following locks held: GenericObjectPool
current access in thread main:
        GenericObjectPool.close(GenericObjectPool.java:888)
        Harness.run(Harness.java:35)
        Harness.main(Harness.java:5)
        sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
        java.lang.reflect.Method.invoke(Unknown Source)
        org.apache.bcel.util.JavaWrapper.runMain(Unknown Source)
        org.apache.bcel.util.JavaWrapper.main(Unknown Source)
        edu.umd.cs.pugh.CheckSync.main(Unknown Source)
 with no locks held
```

Figure 3.8: Continuous data race detection running on Pool. Both images show the top warning. The left screenshot shows the original, buggy Pool implementation; its first warning suggests that the `SleepingObjectFactory.counter` field might have a data race. The developer adds `synchronized` to the method signature (right screenshot) and the top warning disappears without the developer saving the file or invoking data race detection.

3.5.3 Solstice Continuous Testing Usability Evaluation

We evaluated how continuous tools built with Solstice affect developer behavior in two ways. The author of this dissertation used the Solstice continuous testing plug-in (Section 3.5.2) during routine debugging (Section 3.5.3), and we ran a case study (Section 3.5.3).


Option.java

```
public int getId()
{
    return getKey()
        .charAt(0);
}
```

Test Results

Results: (R/P/F/I) 187/187/0/0

Option.java

```
public int getId()
{
    return getKey()
        .charAt(1);
}
```

Test Results

Results: (R/P/F/I) 187/186/1/0

```
Details for failures:
test11458(BugsTest) [String index out of range: 1]:
java.lang.StringIndexOutOfBoundsException:
  String index out of range: 1
    at String.charAt(Unknown Source)
    at Option.getId(Option.java:147)
```

Figure 3.9: Continuous testing running on Apache commons.cli. The left screenshot shows the original commons.cli implementation, for which all tests pass. The developer defines the id of an option to be its second character (right screenshot) and immediately sees that this change causes an existing test to fail, without saving the file or invoking JUnit.

Debugging with Solstice Continuous Testing

The first author used the Solstice continuous testing plug-in (CT$_{\text{Solstice}}$) while debugging a BibTeX management project, consisting of 7 Java KLoC. The project was exhibiting `RuntimeException` crashes on a specific input. The author used $\text{CT}_{\text{Solstice}}$ while writing tests and fixing the defect. This took 3 days, and required an extension to the project's architecture and writing more than 100 LoC.

At the start of the debugging process, the subject program had no tests. The author wrote two tests: a regression test to validate that nothing was broken while fixing the defect, and another test for the failing input to observe the presence of the error. The tests were 60 LoC on average and implemented the following algorithm: parse a bibliography from a hard-coded file, programmatically construct a bibliography that is expected to be equivalent to the parsed one, and assert that two bibliography representations are equivalent. The case study led to the following three observations:

$\text{CT}_{\text{Solstice}}$ can speed up discovering unknown bugs: When an input file did not exist, the program crashed with a `FileNotFoundException`. The author discovered this error early, right after starting implementing the regression test: $\text{CT}_{\text{Solstice}}$ ran an incomplete test with an invalid path. The author would not have thought to run this incomplete test and would have discovered the error later, if at all.

$\text{CT}_{\text{Solstice}}$ simplifies debugging: $\text{CT}_{\text{Solstice}}$ enables live programming [17, 23, 149, 71]. While debugging, developers often use print statements to view intermediate program state and assist in understanding behavior. $\text{CT}_{\text{Solstice}}$ makes the continuous testing console output and error streams available to the developer. With each edit, $\text{CT}_{\text{Solstice}}$ recomputed and redisplayed these logs, giving near-instant feedback on how changes to the code affected the print statements, even if the changes did not affect the test result. The author felt this information significantly simplified the debugging task.

$\text{CT}_{\text{Solstice}}$ is unobtrusive: During this debugging process, the author never experienced a noticeable slowdown in Eclipse's operation and never observed a stale or wrong test result.

Solstice Continuous Testing Case Study

To further investigate how developers interact with Solstice continuous analysis tools, we conducted a case study using the Solstice continuous testing plug-in. This case study investigates the following research questions:

RQ1: What is the perceived overhead for Solstice continuous analysis tools?
RQ2: Do developers like using Solstice continuous analysis tools?

The remainder of the section explains our case study methodology, presents the results, and discusses threats to validity.

	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
Solstice continuous testing					
Test results were always up to date.	1	4	0	0	0
I liked using continuous testing.	2	2	1	0	0

Figure 3.10: Exit survey summary for the user study subjects who used Solstice continuous testing.

Methodology

Each subject implemented a graph library using test-driven development (TDD). The subject was given skeleton `.java` files for the library, containing a complete Javadoc specification and a comprehensive test suite of 93 tests. The method bodies were all empty, other than throwing a `RuntimeException` to indicate that they have not been implemented. Accordingly, all tests failed initially. The subject's task was to implement the library according to the specification and to make all tests pass. The subjects were asked not to change the specification and not to change, add, or remove tests, but they could configure Eclipse as they wished and could use the Internet throughout the task.

For the case study, we recruited 10 graduate students at the University of Washington who were unfamiliar with our research[3]. Half of these subjects were randomly assigned to use JUnit (base treatment) and the remaining half were assigned to use the Solstice Continuous Testing plug-in. Subjects had varying Java (1 to 12 years), JUnit (none to 4 years), and TDD (none to 4 years) experience.

All sessions were conducted in a computer lab[4] at the University of Washington. After a 5-minute introduction that explained the purpose of the study, each subject completed a tutorial to learn the tool they would be using during the session. Then, each subject implemented as much of the graph library as possible within 60 minutes. We recorded the computer screen and snapshotted the subject's codebase each time it was compiled. Finally, we conducted a written exit survey, asking the subjects about their experience.

[3]Subjects were recruited through a standard IRB-approved process. Participation in the study was compensated with a $20 gift card.

[4]Computer specs: Intel i5-750: 2.67 GHz quad core CPU, network drive, 4 GB memory, connected to a 30-inch display.

Results

The test suite executed in under one second (unless the subject implemented methods that took unreasonably long). This short test suite execution time is appropriate for a small library and allowed us to answer **RQ1**; a long-running test suite would have masked the tool's overhead. All subjects agreed that the continuous testing results were always up to date (Figure 3.10). In addition to the Likert-scale questions in Figure 3.10, the exit survey also had a free-from question that asked the subjects to comment on their experience with the Solstice continuous testing plug-in. For example, one subject commented: "I really liked the fast feedback [from continuous testing]." Although the computers were running screen-recording software, two Eclipse instances, and a web browser, when asked during the exit survey if test results had been up to date, all five of the subjects agreed the results were up to date and none complained about lag nor any other evidence of overhead.

During the case study, we observed how developers interacted with Solstice continuous testing. After the tutorial, three developers (out of five) started using the tool as we expected, by repeating the following steps:

1. select a failure from the `Test Failures` view,
2. investigate the corresponding trace in the `Trace` view and navigate to the code locations using hyperlinks,
3. make the required code changes, and
4. verify that the failure is fixed (or discover that it is not) by looking at the updated results in the `Test Failures` view.

One subject had issues with using the two different views: she switched from the `Trace` view to the `Javadoc` view, forgot to switch back, and was confused by not being able to see the trace for the selected test failure. The last subject simply ignored the whole workflow as he was not used to using tools that provide continuous feedback. Figure 3.10 shows that all but one of the subjects liked using Solstice continuous testing. One subject commented: "I really enjoyed [using Solstice continuous testing]! ... [Getting continuous feedback] in a real language like Java was pretty cool."

In addition to our qualitative results, we analyzed the recorded development history of each subject. 84 test failures were fixed by at least one developer from each treatment group. The Solstice continuous testing group fixed 52 of these failures faster, whereas the JUnit group fixed 38 faster. On average, JUnit subjects fixed 62 whereas Solstice continuous testing subjects fixed 49 test failures. As the size of our study was small, none of these results is statistically significant (all $p > 0.05$) according to the Mann-Whitney U test.

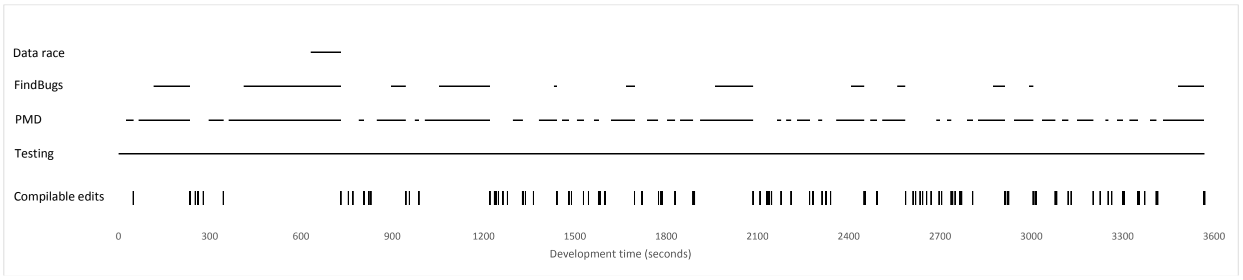


Figure 3.11: Availability of Solstice analyses for one of the case study participants. The x-axis represents the development time in seconds. The vertical lines represent developer edits that yielded compilable code. Solid lines on analyses rows represent the times that the corresponding analysis would have shown up-to-date results during development. (Figure 3.12 summarizes this data across all participants.)

In answering **RQ2**, we conclude that Solstice continuous analyses tools are easy to use, intuitive, and unobtrusive. While our small-scale study has not shown directly the benefits of continuous analysis tools, previous research has done so [71, 87, 128, 65], and reverifying this claim is outside the scope of this work.

Availability of Results for Long-Running Analyses

Our case study used a fast analysis: the time to execute the test suite was under a second. However, executing continuously is beneficial for all analyses, even long-running ones. Executing continuously reduces the cognitive load, since the developer neither has to decide when to run the analysis nor predict when there will be a long enough break in activity to complete the analysis. The continuous analysis eliminates or reduces wait time when the developer desires the analysis results. As discussed in Section 3.2.3, even potentially-stale analysis results have value. Thus, every analysis should be run continuously, under reasonable assumptions: the analysis process is run at low priority to avoid slowing down the developer's IDE, electrical power is less costly than the developer's time, and the UI that presents the results is non-obtrusive.

Given that any amount of increased availability of analysis results is beneficial, we ask how often would those analysis results be available. This section investigates our case study data in a quantitative experiment to estimate the availability of results from long-running Solstice continuous analyses. We focus on the following research question:

RQ3: How does the run time of an A_o affect the availability of its results to the corresponding A_c?

Using the development history (snapshots) of case study participants, we computed the percent availability and the average staleness of each of the four Solstice analyses of Section 3.5.2. Percent availability is the ratio of the total time the analysis results are up to date to the the total development time. Average staleness is the average value for how stale the currently-displayed results are (how long it has been since they were up to date), where up-to-date results are treated as 0 seconds stale.

We assume immediately-interrupting and display-stale (recall Section 3.2) implementations. The number of development snapshots is equal to the number of edits that yielded a compilable project. The analysis results become up to date if the developer pauses longer than $\varepsilon_a + T_{A_o}$, where:

Analysis	A_o Run time	Availability	Avg. Staleness
Testing	0.01 s.	99.7%	0.003 s.
PMD	25 s.	61.2%	10.9 s.
FindBugs	74 s.	33.7%	108.6 s.
Data race	293 s.	5.7%	1124 s.

Figure 3.12: A_o run time, percent availability, and average staleness of each Solstice continuous analysis, averaged over all case study participant data. The results suggest that even a continuous long-running analysis can provide value during development.

ε_a: the continuous analysis result delay (Definition 5).

T_{A_o}: underlying A_o run time.

ε_a and T_{A_o} values are taken from Figure 3.5. The analysis results become stale immediately at the beginning of the next snapshot.

Figure 3.11 shows the developer edits and the availability of each analysis as a timeline, for one of the case study participants. We did this computation for each case study participant. We provide similar figures for the other participants, and the raw data at `https://bitbucket.org/kivancmuslu/solstice/downloads/analysis_availability.zip`.

Figure 3.12 shows percent availability and average staleness of each Solstice analysis, averaged over all case study participants. Although the run time of an A_o has a negative effect on the availability of the corresponding A_c, long-running Solstice analyses would still be beneficial during development. Data race detection results are up to date 5% of the time.

Threats to Validity

We assess our evaluation activities in terms of simple characterizations of internal and external validity. Internal validity refers to the completeness and the correctness of the data collected through the case studies. External validity refers to the generalizability of our results to other settings.

As in other research, the possibility of a defect in the tools is a threat to internal validity. Seeing incorrect information could confuse and slow down developers. However, we received no negative feedback about correctness.

The selection of the subject program, a simple graph library, poses a threat to external validity.

Case study results for this data structure may not generalize to other software.

The selection of the offline analysis, testing, poses another threat to external validity. Case study results on how developers interact with continuous testing may not generalize to other continuous analysis tools. However, we believe the specific internal offline analysis does not affect the developer's interaction with the continuous analysis tool.

Finally, the fact that all our subjects were PhD students poses another threat to external validity. Case study results for a particular developer population may not generalize to other developer populations. However, none of the subjects knew Solstice continuous testing before the case study and their experiences with JUnit, Eclipse, and Java varied. Most subjects had professional experience through internships in industry.

3.5.4 Alternate Implementation Strategies

There are ways other than maintaining a copy codebase to convert offline analyses into continuous ones. Very fast offline analyses can run in the IDE's UI thread. While technically such an analysis would block the developer, the developer would never notice the blocking because of its speed. Most analyses are not fast enough for this approach to be feasible.

It is possible to reduce the running time of an offline analysis by making it incremental [124]. An incremental code analysis takes as input the analysis result on an earlier snapshot of the code and the edits made since that snapshot. Examples include differential static analyses [93], differential symbolic execution [122], and incremental checking of data structure invariants [136]. When the differences are small, incremental analyses can be significantly faster. With this speed increase, incremental analyses may be used continuously by blocking the developer whenever the analysis runs. Incremental code compilation [88] is one popular incremental, continuous analysis integrated into many IDEs. However, many analyses cannot be made incremental efficiently because small code changes may force these analyses to explore large, distant parts of the code. Further, making an analysis incremental can be challenging, requiring a complete analysis redesign. The process is similar to asking someone to write an efficient, greedy algorithm that solves a problem for which only an inefficient algorithm that requires global information is known.

While many analyses cannot be made incremental or efficient enough to run continuously while blocking the developer, those that can still benefit from being built using Codebase Analysis. An impure analysis is freed from the burden of maintaining a copy codebase, as Codebase Analysis maintains the copy codebase and lets the analyses own it exclusively. Codebase Analysis allows long-running analyses to execute on a recent snapshot and produce results that may be slightly stale, whereas other approaches would not.

Codebase Analysis uses a step-based execution model to execute A_o on the copy codebase. Codebase Analysis could instead use a build tool, such as Apache Maven or Ant, letting an analysis author declare how A_o runs via build files. Although using a build tool might further simplify the A_c implementation, it would also limit A_c to the capabilities of the build tool. Step-based execution permits the analysis author to implement A_c using arbitrary Java code or to define A_c as a one-step analysis that executes a build tool.

It is possible to implement memory-change-triggered continuous analysis tools by combining a file-change continuous analysis framework such as Incremental Project Builders with tools that automatically save changes periodically such as the Smart Save plug-in [137]. However, running an analysis on a separate codebase has additional benefits. First, the developer never experiences any unwanted side-effects, such as crashes or code modifications due to impurity, of the analysis. Second, for longer-running analyses, when there is a conflicting developer edit, Codebase Analysis can let the analysis finish its execution on the copy codebase and produce correct — albeit potentially stale — results.

3.6 Contributions

While useful to developers, continuous analyses are rare because building them is difficult. We classified the major design decisions in building continuous analysis tools, and identified the major challenges of building continuous analyses as *isolation* and *currency*. We designed Codebase Analysis, which solves these challenges by maintaining an in-sync copy of the developer's code and giving continuous analyses exclusive access to this copy codebase. We further introduced a step-based execution model that improves Codebase Analysis's currency. We have built Solstice, a Codebase Analysis prototype for Eclipse, and used it to build four open-source, publicly-available continuous analysis Eclipse plug-ins. We have used these plug-ins to evaluate Codebase Analysis's effectiveness and usability.

We have evaluated Codebase Analysis (1) on performance benchmarks, showing that Solstice-based tools have negligible overhead and have access to the up-to-date code with less than 2.5 milliseconds delay, (2) by building continuous analysis tools, demonstrating that Codebase Analysis and Solstice can be used for a variety of continuous tools including testing, heuristic defect finding, and data race detection and that the effort necessary to build new continuous analysis tools is low (each tool required on average 710 LoC and 20 hours of implementation effort), and (3) with case studies with developers that show that Solstice-based tools are intuitive and easy-to-use.

Codebase Analysis provides a simple alternative to redesigning offline analysis logic to work continuously. Overall, the cost of converting an offline analysis to a continuous one with Codebase Analysis is low. Further, the benefits of continuous analysis tools greatly outweigh the cost of building them with Codebase Analysis. We believe that Codebase Analysis, and our implementation, will enable developers to quickly and easily build continuous tools, and will greatly increase the availability of such tools to developers. These tools will reduce the interruptions developers face and the delay before developers learn the effects of their changes, and consequently will positively impact software quality and the developer experience.

Chapter 4

SPECULATIVE ANALYSIS: BRINGING KNOWLEDGE OF FUTURE

Developers interact with tools to automate or simplify common tasks. For example, version control systems (VCSs) simplify collaboration with other developers and integrated development environments (IDEs), such as Eclipse and Visual Studio, provide tools for refactoring, auto-complete, and correction of compilation errors. These tools have two goals: increasing developer speed and reducing developer mistakes. These tools are widely used: they are the most frequent developer actions, provided by the JDT plug-in, after common text editing commands such as delete, save, and paste [112].

Unfortunately, users of these tools are provided with little or no information about their consequences. For example, if a developer wants to understand whether her recent changes conflict with another developer's uncommitted changes, she might commit her changes to a temporary branch and ask the other developer to attempt merging these changes with her local copy. As another example, whenever there is a compilation error in an Eclipse project, Eclipse offers *Quick Fix* proposals: transformations that may resolve the error. However, some of these proposals may not resolve the compilation error and may even introduce new errors. When this happens, a developer may waste time undoing the proposal or trying other proposals, and may even give up on Quick Fix.

Although the previous chapter focused on continuous analysis of developers' current codebase, Codebase Analysis (Chapter 3) supports impure analyses that run on variants of the current codebase (*cf.* Section 3.3.1). This chapter focuses on speculative analysis [19], which explores potential future states of a program to compute the consequences of likely developer actions and presents the pre-computed consequences to the developer when the developer starts doing the corresponding action. A speculative analysis generates likely future states of a codebase by applying potential developer actions, and evaluates and compares these likely future states based on a metric. Codebase Analysis supports *all* speculative analyses since any speculative analysis can be formulated as an impure analysis.

The rest of this chapter applies speculative analysis in the context of Eclipse Quick Fixes.

Its aim is to improve Quick Fix by informing developers of the consequences of each proposal, specifically of the proposal's effect on the number of compilation errors. As a proof-of-concept, we have built an Eclipse plug-in, Quick Fix Scout, that computes which compilation errors are resolved by each proposal.[1] When a user invokes Quick Fix, Quick Fix Scout augments the standard dialog with additional, relevant proposals, and sorts the proposals with respect to the number of compilation errors they resolve.

This chapter makes the following contributions:

- A novel application of speculative analysis for automatically computing the consequences of Quick Fix recommendations.
- An open-source, publicly-available tool — Quick Fix Scout: `http://quick-fix-scout.googlecode.com` — that communicates the consequences of a Quick Fix proposal to the developer.
- A case study that shows that 93% of the time developers apply one of the top three proposals in the reordered dialog (Section 4.4.1).
- A controlled experiment with 20 users that demonstrates that Quick Fix Scout allows developers to remove compilation errors 10% faster, compared to using traditional Quick Fix (Section 4.4.2).

The rest of the chapter is organized as follows. Section 4.1 explains the problem with Eclipse's Quick Fix. Section 4.2 presents the design of the speculative analysis for Eclipse Quick Fixes and presents our tool, Quick Fix Scout. Section 4.3 introduces global best proposals — the additional proposals Quick Fix Scout adds to the dialog. Section 4.4 details the case study and controlled experiment design and results, and threats to the validity of these results. Finally, Section 4.5 concludes with the contributions.

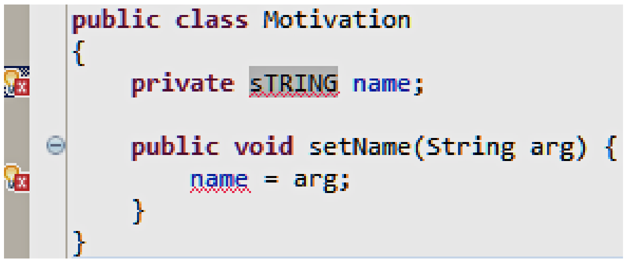
4.1 Not Knowing the Consequences

Eclipse uses a fast, incremental compiler to identify and underline compilation errors with a "red squiggly". A developer who invokes Quick Fix at an error sees a pop-up dialog with a list of actions each of which may fix the error. The Eclipse documentation notes that Quick Fix can be used not only to provide suggestions but also as a shortcut for more expert users. [2]

[1] Although the speculative-analysis-related concepts introduced in this chapter can utilize Codebase Analysis, Quick Fix Scout does not utilize Solstice since this work was done before Codebase Analysis. For coherence, the rest of this chapter is written as if Quick Fix Scout was implemented utilizing Solstice.

[2] `http://wiki.eclipse.org/FAQ_What_is_a_Quick_Fix%3F`

Figures 4.1–4.2 demonstrate a situation in which Quick Fix falls short. Figure 4.1 shows a program with two compilation errors due to a single type mismatch between a variable declaration and a use site. The variable `name` should be declared to be of type `String` but is instead declared as `sTRING`. Invoking Quick Fix at the declaration error shows 12 proposals (the left screenshot of Figure 4.2). The correct proposal — Change to '`String`' — is the fourth choice in the list. Ideally, Eclipse would provide the correct proposal as the first recommendation. Lower positions in the list likely cause the user to spend more time studying the choices or to cancel Quick Fix and address the error manually.



```
public class Motivation
{
    private sTRING name;

    public void setName(String arg) {
        name = arg;
    }
}
```

Figure 4.1: A Java program with two compilation errors. There is only one logical error: the type `sTRING` should be `String`.

Invoking Quick Fix at the use-site of the error is even worse for the developer. The right screenshot of Figure 4.2 shows the 15 Quick Fix proposals, *none* of which resolves either error. Sophisticated users may realize this, cancel the invocation, and finish the change manually. Others may apply a proposal and either quickly realize that this was a poor choice and undo it, or perform more actions attempting to resolve the error, creating successively more difficult situations to recover from.

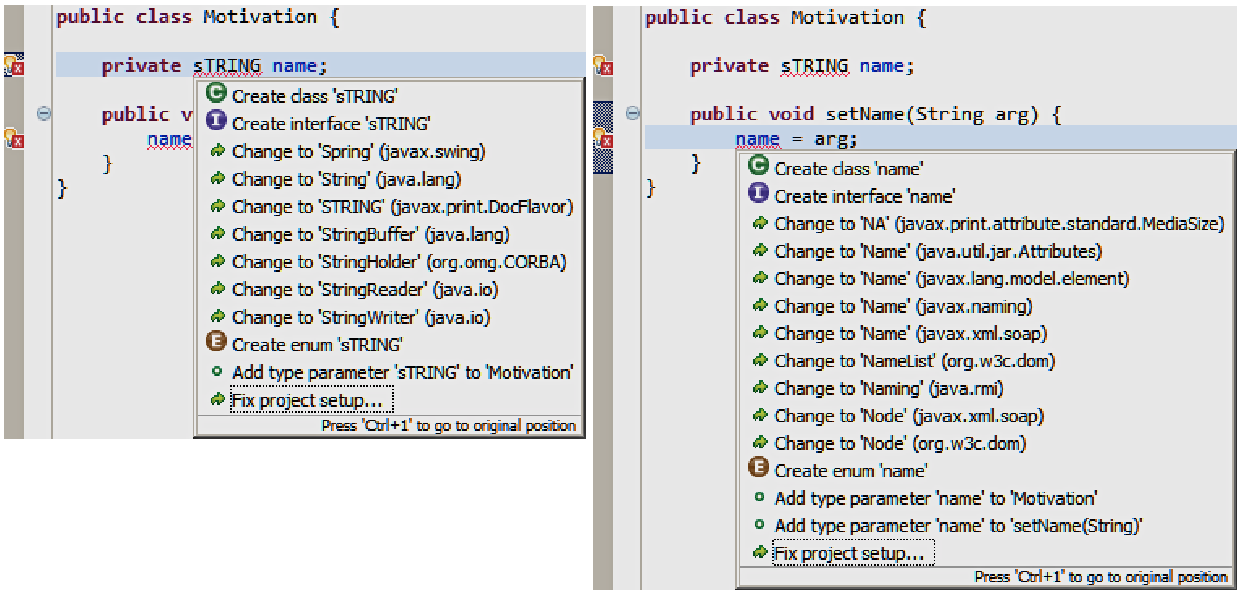
4.1.1 Visualizing Quick Fix consequences

Quick Fix Scout pre-computes the consequences of each proposal and visualizes this information by augmenting the Quick Fix dialog in three ways:

1. To the left of each proposal, add the number of compilation errors that remain after the proposal's hypothetical application.
2. Sort the proposals with respect to the number of remaining compilation errors.
3. Color the proposals: green for proposals that reduce the number of compilation errors, black for proposals that do not change the number of compilation errors, and red for proposals that increase the number of compilation errors.

Figure 4.3 — the Quick Fix Scout equivalent of the left screenshot of Figure 4.2 — shows all these augmentations, except the red coloring.

Section 4.3 discusses one additional feature of Quick Fix Scout: *global best proposal*, which addresses the problem in the right screenshot of Figure 4.2. Changing `sTRING` to `String` (offered only at the first error's site) resolves both compilation errors. However, Quick Fix does not present



```
public class Motivation {

    private sTRING name;

    public v
        name
    }
}
```

Create class 'sTRING'
Create interface 'sTRING'
Change to 'Spring' (javax.swing)
Change to 'String' (java.lang)
Change to 'STRING' (javax.print.DocFlavor)
Change to 'StringBuffer' (java.lang)
Change to 'StringHolder' (org.omg.CORBA)
Change to 'StringReader' (java.io)
Change to 'StringWriter' (java.io)
Create enum 'sTRING'
Add type parameter 'sTRING' to 'Motivation'
Fix project setup...
Press 'Ctrl+1' to go to original position

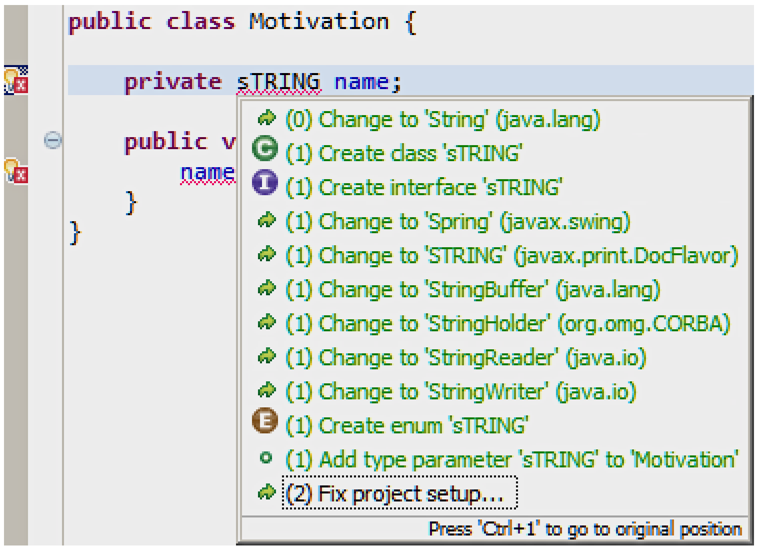
```
public class Motivation {

    private sTRING name;

    public void setName(String arg) {
        name = arg;
    }
}
```

Create class 'name'
Create interface 'name'
Change to 'NA' (javax.print.attribute.standard.MediaSize)
Change to 'Name' (java.util.jar.Attributes)
Change to 'Name' (javax.lang.model.element)
Change to 'Name' (javax.naming)
Change to 'Name' (javax.xml.soap)
Change to 'NameList' (org.w3c.dom)
Change to 'Naming' (java.rmi)
Change to 'Node' (javax.xml.soap)
Change to 'Node' (org.w3c.dom)
Create enum 'name'
Add type parameter 'name' to 'Motivation'
Add type parameter 'name' to 'setName(String)'
Fix project setup...
Press 'Ctrl+1' to go to original position

Figure 4.2: Eclipse offers 12 Quick Fix proposals to resolve the type error (left) and 15 Quick Fix proposals to resolve the assignment error (right) from Figure 4.1. None of the proposals offered for the assignment error resolves either compilation error.

this proposal at the second error's site, even though it is relevant. Quick Fix Scout addresses this problem by providing the relevant proposal at both compilation error locations.

Quick Fix Scout reorders and colors the proposals based on the number of remaining compilation errors. Since the primary intent of Quick Fix is to resolve compilation errors, we assume that a proposal that resolves more compilation errors is likely to be preferred by the developer. The proposals that resolve the same number of compilation errors are sorted using Quick Fix's standard ordering. This allowed us to measure the effects of the main criterion more accurately when evaluating Quick Fix Scout. The empirical data support the premise that developers prefer proposals that resolve the



```
public class Motivation {

    private sTRING name;

    public v
        name
    }
}
```

(0) Change to 'String' (java.lang)
(1) Create class 'sTRING'
(1) Create interface 'sTRING'
(1) Change to 'Spring' (javax.swing)
(1) Change to 'STRING' (javax.print.DocFlavor)
(1) Change to 'StringBuffer' (java.lang)
(1) Change to 'StringHolder' (org.omg.CORBA)
(1) Change to 'StringReader' (java.io)
(1) Change to 'StringWriter' (java.io)
(1) Create enum 'sTRING'
(1) Add type parameter 'sTRING' to 'Motivation'
(2) Fix project setup...
Press 'Ctrl+1' to go to original position

Figure 4.3: Quick Fix Scout sorts the 12 proposals offered by Eclipse (shown in Figure 4.2) by the number of errors that the proposal fixes.

highest number of compilation errors. Case study participants (Section 4.4.1) who used Quick Fix Scout selected a proposal that resolved the most compilation errors 90% of the time. Similarly, controlled experiment participants (Section 4.4.2) who used Quick Fix Scout selected such proposals 87% of the time, and those who used Quick Fix, 73% of the time.

4.2 Quick Fix Scout: Speculative Analysis of Eclipse Quick Fixes

Quick Fix Scout [107] is an Eclipse plug-in that speculatively applies each available Quick Fix proposal and compiles the resulting program. Quick Fix Scout augments the Quick Fix dialog to show how many compilation errors would remain after each proposal's hypothetical application, and sorts the proposals accordingly.

Next, Section 4.2.1 details the mechanism for computing Quick Fix proposals' consequences. Section 4.2.2 explains additional optimizations specific to Quick Fix Scout. Section 4.2.3 discusses implementation limitations. Finally, Section 4.2.4 provides insight into generalizing the technique and the implementation to other IDEs and recommendations.

4.2.1 Computing Quick Fix consequences

Quick Fix Scout uses the speculative analysis algorithm, described at a high level in Figure 4.4, to compute the consequences of Quick Fix proposals. Implemented as an impure Solstice analysis, Quick Fix Scout uses the copy codebase maintained by Codebase Replication. Whenever the developer introduces a new compilation error or fixes an old one (line 2), Quick Fix Scout applies each proposal to the copy (line 5), one at a time, saves and builds the copy (line 6), and associates that proposal with the set of compilation errors that remain (line 7). Quick Fix Scout then undoes the proposal to restore the copy's state (line 8). Quick Fix Scout updates the Quick Fix dialog after computing the consequences of all the proposals (line 10).

4.2.2 Optimizations for a responsive UI

Ideally, Quick Fix Scout computes the consequences of a new error's proposals in the time between when the developer introduces the error and invokes Quick Fix. Quick Fix Scout includes the following optimizations and heuristics:

- It only recomputes consequences if a code change affects the compilation errors, as described in Figure 4.4.

```
while (true) {
  waitUntilChangeInErrors();
  for (Error err: copy.getErrors()) {
    for (Proposal p: err.quickFixes()) {
      copy.applyProposal(p);
      copy.saveAndBuild();
      results.add(p, copy.getErrors());
      copy.applyProposal(p.getUndo());
    }
    publishResults();
  }
}
```

Figure 4.4: A high-level description of the speculative analysis algorithm for computing the compilation errors that remain after applying each Quick Fix proposal. The publishResults() method augments the Quick Fix dialog with the proposal consequences.

- It uses a user-adjustable typing session length to identify atomic sets of changes. A series of edits without a typing-session-length pause constitute an atomic set of edits. Quick Fix Scout waits for an entire atomic session to complete before recomputing consequences. Thus, for example, Quick Fix Scout ignores the temporary compilation errors that arise in the middle of typing a complete token.
- It considers first the errors that are closest to the cursor in the currently open file.
- It caches the consequences (i.e., the remaining compilation errors) for each proposal and uses the cache whenever Eclipse offers the same proposal at multiple locations.
- It updates the Quick Fix dialog incrementally, as results for errors (but not individual proposals for each error) become available. This is shown in Figure 4.4.

In general, each proposal application is a small change and, even for large projects, Eclipse can incrementally compile the updated project extremely quickly. Therefore, Quick Fix Scout's computation scales linearly with the number of proposals (which is proportional to the number of compilation errors), and independently of the size of the project. During typical coding, at any particular time, a project has several compilation errors with several proposals for each. The total number of proposals is typically less than a couple hundreds. As a worst-case example, we experimented with an 8K-line project with 50 compilation errors and 2,400 proposals. A 2.4GHz Intel Core i5 (quad core) MacBook Pro with 8GB of RAM computed all the consequences in 10 seconds, on average (computed over 10 consecutive computations, after allowing Eclipse's incremental compiler to optimize). This suggests Quick Fix Scout can scale well to large projects.

Finally, since each proposal is analyzed separately, the analysis can be parallelized, though we have not yet implemented that functionality.

4.2.3 Current implementation limitations

There are at least four ways to invoke Quick Fix in Eclipse: (1) by pressing the keyboard shortcut, (2) by selecting Quick Fix through the context menu, (3) by clicking on the icon on the left of the screen, and (4) by hovering the mouse over the compilation error. Internally, the first three methods create a Quick Fix dialog and the last method creates a Hover Dialog. The Hover Dialog is handled by `org.eclipse.jdt.ui` plug-in and the Eclipse installation does not permit us to modify this plug-in as we modified `org.eclipse.jface.text`. Though we have an implementation that works in debug mode for the Hover Dialog, our installation fails when it includes a modified `jdt.ui`. A future version of Eclipse will include a public API for reordering content assist type recommendations (e.g., auto-complete and Quick Fix),[3] which would simplify our implementation and might remove this limitation.

For each proposal, the Eclipse API provides an *undo change* that rolls back the associated proposal application. After analyzing each proposal, Quick Fix Scout uses this mechanism to return the copy project to its initial state. The proposals "Change compilation unit to '`typeName`'" and "Move '`typeName`' to '`packageName`'" have a defect in their implementation: the corresponding undos do not restore the project to its original state.[4] We have reported both bugs to Eclipse and they have been reproduced by the developers, but they have not yet been resolved. Quick Fix Scout must either skip analyzing these two proposals or re-copy the copy project after their analysis. Since re-copying can take considerable time for large projects, for performance reasons, the current implementation skips the analysis of these proposals and produces no consequence information for them, leaving the appropriate lines in the Quick Fix dialog unaugmented.

Quick Fix Scout uses an internal Eclipse API to apply proposals to the copy project. By default, this API acts as a no-op for the proposals that require user interaction. Therefore, currently, Quick Fix Scout does not compute the consequences of these proposals and leaves the appropriate lines in the Quick Fix dialog unaugmented. However, to our best knowledge, there are only four such proposals: Create class, interface, annotation, and enum '`typeName`'. These proposals do not modify existing code, but instead create new code. Therefore, it is relatively simple for developers

[3] `http://blog.deepakazad.com/2012/03/jdt-3842-m6-new-and-noteworthy.html`

[4] `https://bugs.eclipse.org/bugs/show_bug.cgi?id=338983` and `https://bugs.eclipse.org/bugs/show_bug.cgi?id=339181`

to mentally predict their consequences.

4.2.4 Generalizing beyond Quick Fix

The ideas we demonstrated on Quick Fix Scout within Eclipse also apply to engines that produce other types of recommendation, such as refactorings and automatic code completions, and to other IDEs, such as NetBeans, IntelliJ, and Visual Studio.

Any recommendation may become obsolete when the code changes. Thus, most of the optimizations and heuristics in Section 4.2.2 apply to other recommendations. For example, automatic code completions that are closest to the current cursor position can be prioritized and computed first.

Finally, Quick Fix Scout is an instantiation of speculative analysis: the future states are generated via Quick Fix proposals, and the consequences are represented by the number of remaining compilation errors. By generating future states and representing consequences in other ways, speculative analysis can generalize to other consequences and recommendation engines. For example, refactoring suggestions can generate future states, and failing tests could represent the consequences.

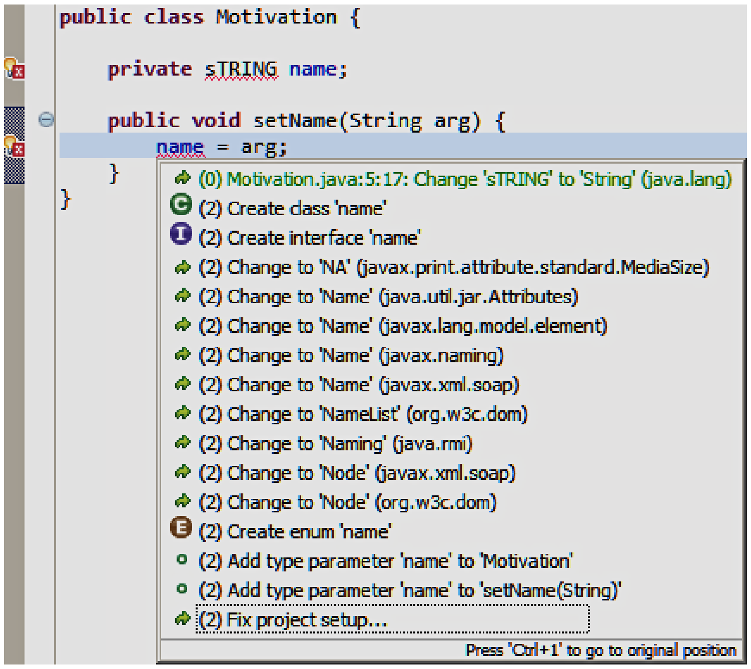
4.3 Global Best Quick Fixes

Quick Fix Scout helps developers to quickly locate the best local proposals — the proposals that resolve the most compilation errors — by sorting them to the top in the Quick Fix dialog. However, sometimes, Eclipse offers the best proposal to fix an error at a *different* location than the error itself (recall Section 4.1). Quick Fix Scout's speculative analysis handles such situations because the analysis is global and applies to all compilation errors and proposals in the project, thus computing the information necessary to offer the global best proposal at all the relevant locations [108]. Figure 4.5 (the Quick Fix Scout equivalent of the right screenshot of Figure 4.2) shows a global best proposal at the top of the dialog. That proposal is suggested by Eclipse at a different compilation error location, and is not displayed by the original Quick Fix.

For global best proposals, Quick Fix Scout adds the following context information:

1. The location of the error where Eclipse offers the proposal (`Motivation.java:5:17` in Figure 4.5).
2. The remote context that will be modified (`'sTRING'` is added to the original message `Change to 'String'` in Figure 4.5).

While this context information is not necessary for local proposals, it is useful when the proposal is displayed at a different location than the error to which it directly applies. For example, a developer may interpret `Change to 'String'` incorrectly, without knowing what token, and on what line, will be changed to `'String'`.



```
public class Motivation {

    private sTRING name;

    public void setName(String arg) {
        name = arg;
    }
}
```

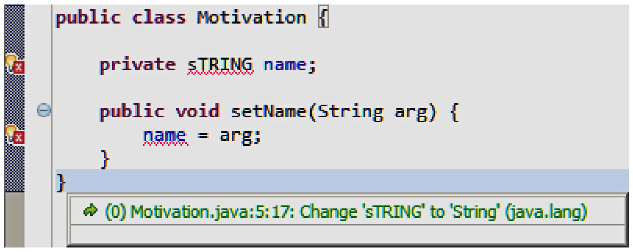
(0) Motivation.java:5:17: Change 'sTRING' to 'String' (java.lang)
(2) Create class 'name'
(2) Create interface 'name'
(2) Change to 'NA' (javax.print.attribute.standard.MediaSize)
(2) Change to 'Name' (java.util.jar.Attributes)
(2) Change to 'Name' (javax.lang.model.element)
(2) Change to 'Name' (javax.naming)
(2) Change to 'Name' (javax.xml.soap)
(2) Change to 'NameList' (org.w3c.dom)
(2) Change to 'Naming' (java.rmi)
(2) Change to 'Node' (javax.xml.soap)
(2) Change to 'Node' (org.w3c.dom)
(2) Create enum 'name'
(2) Add type parameter 'name' to 'Motivation'
(2) Add type parameter 'name' to 'setName(String)'
(2) Fix project setup...

Figure 4.5: Quick Fix Scout computes the global best proposal for each compilation error and adds it to the Quick Fix dialog for that error. The addition of the associated error location (`Motivation.java:5:17`) and the associated error context (`'sTRING'`) distinguish global best proposals from normal proposals. If the global best proposal is already one of the local proposals, Quick Fix Scout makes no additions.

As a consequence of the above process, global best proposals are only shown if they resolve the local error, among other errors. While it is possible to augment the dialogs of all errors with the proposal that resolves the most errors in the project overall, we believe that showing a fix for an unrelated error might confuse developers. However, if invoked on a location without a warning or a compilation error, Quick Fix Scout does show the proposal that resolves the most errors (Figure 4.6).

One of the controlled experiment (Section 4.4.2) participants articulated the usefulness of global best proposals:

> "[Global best proposals] were great, because honestly the source of error is often not at the [location where I invoke Quick Fix]."



```
public class Motivation {

    private sTRING name;

    public void setName(String arg) {
        name = arg;
    }
}
```

(0) Motivation.java:5:17: Change 'sTRING' to 'String' (java.lang)

Figure 4.6: If invoked on a location without a warning or a compilation error, Quick Fix Scout shows the proposals that resolve the most errors whereas the default implementation would only inform the user that there are no available proposals for that location.

4.4 Evaluation

Our evaluation was based on two activities. First, over a roughly one-year period, we distributed a version of Quick Fix Scout to a collection of 13 friendly users (including the project team) and gathered information about their Quick Fix and Quick Fix Scout behavior in their normal workflow (Section 4.4.1). Second, we ran a controlled experiment with a within-participants mixed design across 20 participants, asking them to resolve various compilation errors on code they had not previously seen (Section 4.4.2).

The friendly users selected, at their discretion, to use either Quick Fix or Quick Fix Scout during each logged session. The design of the controlled experiment determined the situations in which participants used Quick Fix and Quick Fix Scout.

For both activities, we acquired data with an instrumented version of the tool. The tool logs:

- whether Quick Fix or Quick Fix Scout is is running,
- the proposals offered by Quick Fix or Quick Fix Scout,
- whether the user selected a Quick Fix proposal or canceled the invocation,
- which proposal the user selected, if any, and
- how long it took the user to either make a selection or cancel the invocation.

The tool also tracks information that lets us detect some situations in which a user applies a proposal but soon after undoes that proposal.

4.4.1 Case study: friendly users

The goal of our informal case study was to understand how Quick Fix is used "in the wild" by developers. We wished to investigate the following questions:

- Does the ordering of the displayed proposals affect which proposal is selected?
- Does the number of proposals displayed affect which proposal is selected?
- Does the kind of proposal displayed affect which proposal is selected?

Case study design

Over approximately one year, 13 developers — including the project team — ran our tool and allowed us to view its logs. For each Eclipse session, each participant was free to use either the standard Quick Fix or our Quick Fix Scout; all sessions were logged.

Case study results

Figure 4.7 shows that users selected the first (top) proposal 70% of the time, one of the top two proposals 90% of the time, and one of the top three proposals 93% of the time. For Quick Fix Scout sessions, the percentages are slightly higher, at 78%, 93%, and 95%. Given the small difference,

	Standard Quick Fix					**Quick Fix Scout**				
User ID	**# completed sessions**	**QF selection rate 1st**	**2nd**	**3rd**	**top 3**	**# completed sessions**	**QF selection rate 1st**	**2nd**	**3rd**	**top 3**
1	4	100%	0%	0%	100%	1	100%	0%	0%	100%
2	0		—			1	100%	0%	0%	100%
3$\star$	45	64%	16%	13%	93%	362	81%	15%	2%	98%
4	167	78%	20%	1%	99%	0		—		
5	17	47%	24%	0%	71%	0		—		
6$\star$	25	40%	24%	8%	72%	22	55%	27%	0%	82%
7$\star$	82	70%	22%	2%	94%	28	68%	18%	0%	86%
8	9	67%	22%	11%	71%	0		—		
9	7	71%	0%	0%	71%	10	60%	10%	10%	80%
10	6	33%	17%	33%	83%	0		—		
11	0		—			0		—		
12	6	17%	0%	17%	34%	0		—		
13	0		—			2	50%	0%	0%	50%
All	368	69%	20%	4%	93%	426	78%	15%	2%	95%

Figure 4.7: Case study information. A $\star$ in the User ID indicates the participant is a collaborator for Quick Fix Scout. Completed sessions are the number of times the user invoked Quick Fix and selected a proposal. For each of the first three proposals in the Quick Fix menu, we report how often that proposal was selected. For example, user 9 never selected the second or third offered proposal from a standard Quick Fix menu, but did so when using Quick Fix Scout.

and that three of the participants are from the project team, this data does not confirm a hypothesis that Quick Fix Scout is different from Quick Fix in this dimension.

For the completed sessions, Quick Fix offered as many as 72 (mean=5.7, median=2) proposals. For the canceled sessions, Quick Fix offered as many as 80 (mean=6.4, median=4) proposals. In contrast, for the completed sessions, Quick Fix Scout offered as many as 38 (mean=4.2, median=2) proposals. For the canceled sessions, Quick Fix Scout offered as many as 27 (mean=5.1, median=3) proposals. These data may suggest that when a user does not find an expected or a useful proposal easily, the invocation is more likely to be canceled. To investigate this further, we looked for a correlation between the number of proposals offered in the dialog and the session completion rate. We found no such correlation, further suggesting that as long as the correct proposal is located near the top of the list, the number of proposals shown might not have an effect on developers' decisions.

Eclipse documentation categorizes the Quick Fixes (see the column headings in Figure 4.8).[5]

[5]`http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/ref-java-editor-quickfix.htm`

User ID	Types	Exception Handling	Methods	Constructors	Fields & Variables	Other	Unknown	Package Declaration	Imports	Build Path Problems
1	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
2	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
3	26%	22%	29%	11%	9%	1%	3%	0%	0%	0%
4	2%	65%	11%	10%	1%	5%	2%	0%	4%	0%
5	94%	0%	0%	0%	6%	0%	0%	0%	0%	0%
6	51%	19%	17%	0%	2%	0%	6%	0%	4%	0%
7	49%	0%	13%	9%	14%	7%	3%	5%	0%	1%
8	44%	0%	0%	11%	33%	11%	0%	0%	0%	0%
9	59%	18%	6%	6%	12%	0%	0%	0%	0%	0%
10	67%	0%	0%	0%	33%	0%	0%	0%	0%	0%
11	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
12	0%	0%	83%	0%	17%	0%	0%	0%	0%	0%
13	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
All	30%	27%	21%	10%	8%	3%	3%	3%	1%	1%

Figure 4.8: Proposal types and their selection ratios during the case study. The proposals whose type was unclear are listed as "Unknown".

Five out of the nine proposal types represent 92% of all selected proposal types.

Figure 4.9 presents the most-frequently selected proposals and their selection ratios. Except for one user, these six proposals constitute about 80% of the selected proposals. Note the similarity in selection ratio between the proposals "Import . . . ", "Add Throws Declaration", and "Add Unimplemented Methods" and their types "Types", "Exception Handling", and "Constructor" respectively. The "Change to . . . " proposal falls into "Methods" and "Fields & Variable", depending on its recipient.

Though there is some variation between participants, the results suggest that all proposals *do not* have the same importance: there are a few proposals that are favored by the developers. This observation can be explained by the nature of these proposals. For example, the "Import . . . " proposal is offered whenever the developer declares an unresolvable type. If the developer makes this mistake intentionally, most of the time, she either wants to import that type or create a new type with that name. Therefore, there is a high probability that one of these proposal will be selected. "Add throws declaration" and "Surround with Try/Catch" are two proposals that are always offered for exception-handling-related compilation errors. When there is an exception-handling error, it is very likely that the developer will either propagate that exception or handle it immediately, which suggests that one of these proposals will be selected.

The imbalance in proposal selection rate can be used to improve Quick Fix by prioritizing proposals with respect to the user's history. Bruch et al. [18] have already done this for auto-complete.

User ID	Import ...	Add Throws Declaration	Create Method ...	Change to ...	Add Unimplemented Methods	Surround with Try/Catch	Total
1	100%	0%	0%	0%	0%	0%	100%
2	100%	0%	0%	0%	0%	0%	100%
3	24%	21%	21%	10%	7%	0%	83%
4	2%	47%	11%	1%	8%	14%	83%
5	76%	0%	0%	6%	0%	0%	82%
6	34%	11%	2%	26%	0%	6%	79%
7	37%	0%	11%	7%	9%	0%	64%
8	44%	0%	0%	33%	11%	0%	88%
9	53%	18%	0%	24%	6%	0%	100%
10	50%	0%	0%	50%	0%	0%	100%
11	0%	0%	0%	0%	0%	0%	0%
12	0%	0%	0%	83%	0%	0%	83%
13	0%	0%	0%	0%	0%	0%	0%
All	25%	23%	15%	10%	7%	4%	84%

Figure 4.9: Most-frequently selected proposals and their selection ratios for the case study. Proposals that are selected less than 3% overall (represented by the "All" row) are excluded.

4.4.2 Controlled experiment: graduate students

The goal of our controlled experiment was to determine whether users behave differently when using Quick Fix and when using Quick Fix Scout.

Each participant performed two sets of tasks — α and β task sets — of 12 tasks each. Each task presented the participant with a program that contained at least two compilation errors and required the participant to resolve all the compilation errors. The non-compilable program states were chosen randomly from the real development snapshots captured during the case studies from Section 4.4.1. For 6 of the tasks in each task set, we manually seeded each task with either 1 or 2 additional mutation errors, such as changing a field type or a method signature. The mutations introduced an average of 2.8 extra compilation errors per task.

Our study answers two research questions:

RQ 1: Does the additional information provided by Quick Fix Scout — specifically, the count of remaining compilation errors, and the coloring and reordering of proposals — allow users to remove compilation errors more quickly?

RQ 2: Does Quick Fix Scout affect the way users choose and use Quick Fix proposals?

Controlled experiment design

We recruited 20 participants, all graduate students who were familiar with Quick Fix but had never used Quick Fix Scout.[6]

[6] Approved human subject materials were used; participants were offered a $20 gift card.

We used a within-participants mixed design. We considered two factors: the tool or treatment factor (Quick Fix vs. Quick Fix Scout), and the task factor (α vs. β task sets). To reduce the confounding effects from developer differences and learning effects, we defined four blocks — the cross-product of the two factors. We used a balanced randomized block protocol, randomly selecting which participants perform which block with a guarantee that each block is performed an equal number of times. (We rejected a full within-participants factorial design because of the learning effects we would anticipate if a participant performed the same set of tasks twice using Quick Fix and then Quick Fix Scout or vice versa.)

Each participant received a brief tutorial about Quick Fix Scout, performed the two blocks (task sets), and took a concluding survey comparing Quick Fix Scout and Quick Fix around the two blocks. The two blocks differed in both the tool/treatment factor (from Quick Fix to Quick Fix Scout, or vice versa) and also the task factor (from the α task set to the β task set, or vice versa).

To answer **RQ 1**, we measured the time it took participants to complete tasks. In addition to the time per task group (α and β), we calculated per-task time by using the screen casts. The beginning of a task is defined to be the time when the participant opens the related project for the first time and the end of a task is defined to be the time when the participant resolved all compilation errors in the task and was satisfied with the implementation.

To answer **RQ 2**, we measured whether the user selected a proposal after invoking the Quick Fix menu or canceled the menu, how long it took the user to make that decision, which proposal the user selected, and whether the user undid a selected proposal.

Controlled experiment results

We used R to perform a 4-way blocked MANOVA test utilizing all independent and dependent variables. This minimizes the risk of a type 1 statistical error. All independent variables (user, Quick Fix vs. Quick Fix Scout, task, and order of task) had statistically significant effects, so we examined the analysis-of-variance results of the MANOVA test.

treatment type		**1st treatment**	**2nd treatment**	**all treatments**
α	QF	27m	19m	23m
	QFS	17m	15m	16m
β	QF	31m	22m	27m
	QFS	36m	21m	29m
QF		29m	20m	25m
QFS		26m	18m	22m

Figure 4.10: Mean time to remove compilation errors, in minutes.

RQ 1 Participants completed tasks 10% faster, on average, when using Quick Fix Scout than Quick Fix (Figure 4.10). However, this result was not statistically significant (p=.11).

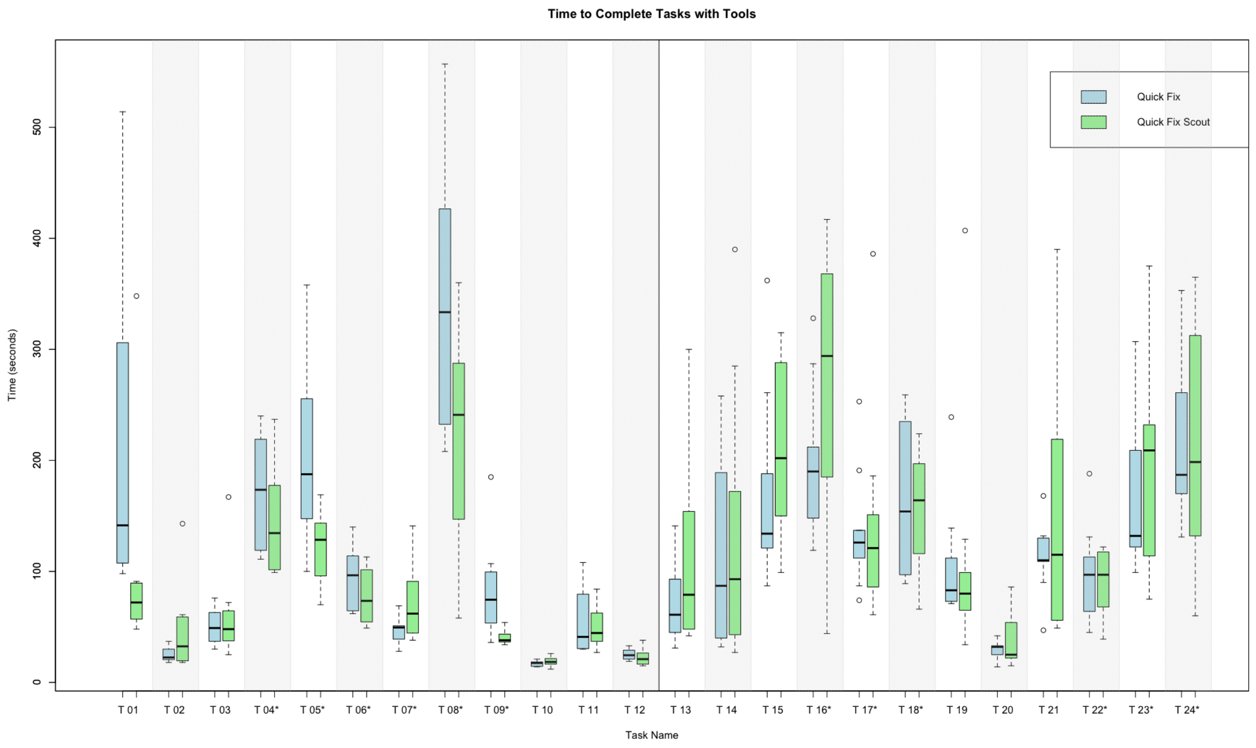


Figure 4.11: Median, minimum, and maximum time spent to complete each task by participants in seconds with and without Quick Fix Scout. The first 12 tasks make up the α set and the last 12 tasks make up the β set. The tasks with seeded errors are followed by an asterisk. Outliers are represented as small circles.

All the other independent variables did have statistically significant effects on task completion time: user (p=5×10^{-7}), task (p=2×10^{-16}), and order (p=3×10^{-6}).

Even the task group had an effect (p=3×10^{-5}): tasks in the β group were harder, and in fact five participants could not complete all tasks in β. We had not anticipated any difference between the task groups. Future work should investigate how the β tasks differ from the α tasks and why Quick Fix Scout caused a slight (but not statistically significant) slowdown on the β tasks. Per-task descriptive statistics appear in Figure 4.11.

There is a learning bias (p=4×10^{-8}): the participants completed a task set 22% faster if it was their second task set. Possible explanations for this bias include participants getting used to resolving compilation errors and participants becoming familiar with the code (since multiple tasks were drawn from the same development projects).

RQ 2 Figure 4.12 summarizes the data we collected regarding user behavior with respect to Quick Fix.

treatment		# invocations (invs.)	undone invs. rate	+invs. rate	avg. time +invs.	avg. time -invs.	1st prop. rate	2nd prop. rate	3rd prop. rate	BP rate	GBP rate
α	QF	554	17%	58%	3.0s	6.8s	79%	18%	0%	76%	
	QFS	449	10%	68%	4.6s	8.8s	79%	16%	0%	90%	79%
β	QF	572	13%	56%	4.3s	7.9s	73%	20%	2%	71%	
	QFS	631	17%	55%	4.4s	6.8s	71%	22%	2%	85%	67%
QF		1116	15%	57%	3.7s	7.4s	76%	19%	1%	73%	
QFS		1080	14%	60%	4.5s	7.4s	75%	19%	1%	87%	75%

Figure 4.12: Quick Fix and Quick Fix Scout invocation duration and proposal selection rate results. Invocations that were immediately undone by the participant are excluded. + and - invocations are ones for which the participant selected and did not select a proposal, respectively. For each treatment, we report the rates with which participants chose the 1st, 2nd, and 3rd proposal, as well as the best (BP) and global best proposals (GBP). Best proposals are defined as the proposals that resolve the highest number of compilation errors for a given Quick Fix invocation.

Question	Both	Quick Fix Scout	Quick Fix	Neither
Quick Fix (Scout) is helpful when resolving compilation errors.	19	1	0	0
There was no performance issues before Quick Fix dialog is updated.	10	1	2	7
For some tasks, I undid a proposal when using Quick Fix (Scout).	12	1	7	0
I manually resolved errors more often with Quick Fix (Scout).	2	0	10	8

Figure 4.13: The four-question survey, and a summary of the participants' responses, administered at the end of each participant's experiment session.

Use of Quick Fix Scout improved the best proposal selection rate by 14% (p=10^{-8}). This increase and the frequent (75%) usage of global best proposals suggest that the participants were resolving more compilation errors per Quick Fix invocation with Quick Fix Scout. Though the difference between the total number of Quick Fix invocations is low (36) between treatments, we believe that the Quick Fix Scout increased usefulness of completed Quick Fix invocations, which helped the participants to save time overall. One participant noted:

> "With [Quick Fix Scout] I had a much better idea of what the error was. . . I found [Quick Fix] to be more vague. . . "

Use of Quick Fix Scout increased by .8 seconds the time spent selecting a proposal (p=.004). Possible explanations include that (1) the Quick Fix Scout dialog contains extra information that the participants took extra time to process, and (2) Quick Fix Scout may take time to compute causing the participants to wait for the information to appear. Explanation (1) also explains the overall productivity improvement. If the participant gets enough information from the dialog, she could resolve the error without having to investigate the related code. Supporting this hypothesis, half of the participants agreed that they needed to type more manually — instead of using Quick

Fix proposals — to resolve compilation errors when not using Quick Fix Scout (Figure 4.13).

Use of Quick Fix Scout had no other statistically significant effects. This stability between treatments strengthens our hypothesis that Quick Fix Scout did not change the way participants used Quick Fix, rather the extra information provided by Quick Fix Scout increased participants' understanding of the code and helped them make better decisions. One participant noted:

> "It was pretty apparent after using regular Quick Fix second, that [Quick Fix] Scout sped things up. I got frustrated as I'd have to scan from error to error to fix a problem rather than just go to the first error I saw. What's more, I had to spend more time staring at the [Quick Fix dialog] often to find that there was nothing relevant."

The data, the analysis, and the qualitative insights from the case study and controlled experiment participants suggest that **RQ 2** holds: Quick Fix Scout indeed changes the way in which users choose and use Quick Fix proposals. We have not teased out which aspects of Quick Fix Scout have the most influence.

4.4.3 Threats to Validity

We assess our evaluation activities in terms of simple characterizations of internal and external validity. Internal validity refers to the completeness and the correctness of the data collected through the experiments. External validity refers to the generalizability of our results to other settings.

One threat to internal validity is that, due to implementation difficulties, we log all Quick Fix invocations *except* those invoked through the Hover Dialog. We tried to limit this threat by rejecting participants who indicated that they consistently use Hover Dialog for invoking Quick Fix and by mentioning this restriction to accepted participants, recommending that they invoke Quick Fix in a different way. So the data we logged about invocations is accurate, although it may be incomplete.

Another threat to internal validity is in our computation of which proposal resolves the most errors. Since the developer might complete a Quick Fix invocation before the speculation computation completes, and because some proposals are omitted *a priori* (for example, a "Create class" proposal), we may not always log the number of compilation errors that would remain for every Quick Fix proposal. In some cases, these omitted proposals could resolve more compilation errors than the ones we identify as resolving the most errors. In our case study, only 6% of all completed Quick Fix invocations are completed before the speculative analysis finishes. Further, in our case study, none of the users selected instances of the *a priori* omitted proposals.

In addition to common external validity threats (such as having students rather than professional developers as participants), a key threat is the decisions we made about which programs to use in the controlled experiment:

- Using small programs with multiple compilation errors.
- Using snapshots from the case study participants to populate our tasks in the controlled experiment.
- Adding seeded errors to half of the snapshots using mutation operators, as well as using a specific set of mutation operators.

Although there are strong motivations for each of these decisions in our experimental design, they could still, in principle, lead to inaccurate conclusions about how Quick Fix Scout would work if it were broadly distributed and used.

4.5 Contributions

Quick Fix Scout is an enhancement of Eclipse's standard Quick Fix that computes and reports to the user the number of compilation errors that would remain in the program for each Quick Fix proposal. Our prototype Eclipse plug-in addresses issues ranging from challenges in the user interface (additional information must be presented in roughly the same space used by the Quick Fix dialog) to challenges in keeping a background copy of the developer's code in sync with the dynamically changing code (Quick Fix Scout uses a copy to speculatively apply the proposals). We evaluated Quick Fix Scout in two ways: an informal case study of how a set of friendly users use both Quick Fix and Quick Fix Scout in their own work, and a 20-user, within-subjects, mixed-design controlled experiment that compares Quick Fix and Quick Fix Scout. Users fixed compilation errors 10% faster, on average, when using Quick Fix Scout, although this improvement was not statistically significant.

The use of speculative analysis in software development is promising but full of technical challenges. Not having access to Codebase Analysis (Chapter 3), the design and the implementation of Quick Fix Scout was over-complicated due to creating and maintaining the copy codebases, and handling concurrent and conflicting developer edits. The final prototype suffers a bug that makes the copy codebases go out-of-sync for an unknown reason, occasionally. We estimate[7] the wasted effort due to these non-essential difficulties as 2,000 (15%) lines of unnecessary code. The difficulties we faced while designing and implementing Quick Fix Scout led to the design of Codebase Replication (Chapter 2) and Codebase Analysis — the core techniques of this dissertation.

[7]We computed the size of the final codebase and investigated the VCS logs.

Chapter 5

CODEBASE MANIPULATION: SIMPLIFYING INFORMATION RETRIEVAL FROM THE HISTORY

Most software development uses version control to enable collaboration and to create a development history. The version control history is useful for many tasks, such as localizing changes that caused regression failures, identifying developers responsible for specific code, and examining recent changes. However, each of these tasks is best performed at a different granularity of history. For example, to localize the cause of a regression, an analysis tool needs access to a history of all compilable revisions. Meanwhile, for manual inspection, the developer may want to see only the changes relevant to a particular method or class.

Today's approaches generate inflexible histories, each of which works well for only a subset of software engineering tasks. Manually-managed histories are coarse-grained and can help a developer overview recent changes; however, manual commits often tangle changes made for multiple development tasks, such as fixing a defect and refactoring code, which makes the history too coarse-grained and poorly suited for searching for the cause of a regression failure. By contrast, automatically-managed approaches record all developer actions [98, 117, 153] and lead to fine-grained histories that are well-suited for studying developer behavior [116, 146], but poorly suited for manual examination.

This chapter presents Codebase Manipulation, a technique for automatically managing *multi-grained* views of a development history. With Codebase Manipulation, a single history is flexible enough to serve many development tasks. Extending the way Codebase Replication (Chapter 2) detects the developer edits and provides corresponding development snapshots by applying these edits to the copy codebase, Codebase Manipulation automatically records a fine-grained development history. On top of this fine-grained history, Codebase Manipulation applies granularity transformations to represent the history in multi-grained views. Codebase Manipulation's granularity transformations improve the effectiveness of manual and automated analyses that rely on the development history.

Codebase Manipulation's granularity transformations are built on three primitive operations:

Collapse: Combine several edits into a single edit.

Expand: Split a previously collapsed edit into its parts.

Swap: Change the order of two edits.

Building on these basic operations, Codebase Manipulation can transform a fine-grained development history into histories of many desirable granularities, such as file-level changes, compilable code, coherent edits, and other custom granularities. Transforming the history into the appropriate granularity improves its utility for certain software engineering tasks.

The main contributions of this chapter are:

- Identifying the problem: modern development history management's *inflexible granularity* poses an obstacle for history-based information retrieval.
- Identifying the collapse, expand, and swap operations as the primitive building blocks of all high-level granularity transformations.
- The design of Codebase Manipulation, a technique to produce multi-grained development history views.
- Bread, an open-source, publicly-available implementation of Codebase Manipulation for the Eclipse IDE: `https://bitbucket.org/kivancmuslu/chronos`.
- Untangler, an analysis that enables developers to craft cohesive commits, each representing a high-level development task. Untangler helps rewrite part of the development history into cohesive commits by applying multiple levels of granularity transformations. Untangler is available as open source: `https://bitbucket.org/LukeSwart/untangler`.
- Bisector, a history bisection algorithm that pinpoints the cause of a regression failure by using the compilable code granularity history, and an open-source implementation: `https://bitbucket.org/kivancmuslu/chronos-test-bisection`.

Section 5.4 shows that Codebase Manipulation effectively creates the multi-grained views necessary for three different development tasks. Section 5.5 shows that our prototype's overhead is negligible.

The rest of this chapter is organized as follows. Section 5.1 formally defines Codebase Manipulation concepts. Section 5.2 details the Codebase Manipulation technique, and Section 5.3 describes our prototype. Section 5.4 evaluates Codebase Manipulation's expressiveness, and Section 5.5 evaluates performance. Finally Section 5.6 summarizes our contributions.

5.1 Definitions

Our goal is to simplify development history information retrieval by automatically recording a single, fine-grained version control history and by providing automated granularity transformations to make the history available at multiple granularities. To aid in understanding Codebase Manipulation's high-level granularity transformations, we first explain how Codebase Manipulation represents the development history and how Codebase Manipulation's primitives operate on that history.

This section defines the representation and primitives, and Section 5.2 details the high-level granularity transformation algorithms. For brevity, the definitions ignore file creation and deletion; they can be extended to handle these actions. The algorithms assume the code is stored in an array, indexed by filepath and an in-file offset, so 's[filepath]' is a character array of the contents of the file filepath, and 's[filepath][4]' is the 5th character of the file filepath. Appending is represented by the + operator.

Definition 7 (Snapshot)**.** A snapshot s is a single developer's view of a program at a point in time, including the current contents of unsaved editor buffers. Unsaved editor buffers have priority: if a file on disk differs from the editor buffer for that file, the snapshot contains the contents of the editor buffer.

An edit can either be atomic or compound. An *atomic edit* encodes how one chuck of text in a file can be replaced with another chuck; either the original or the final chunk of text may be empty. A *compound edit* is a sequence of edits, each of which is either atomic or compound. A *development history* is an edit that can be applied to the empty snapshot, $\emptyset$. And two development histories views of each other if when applied to $\emptyset$, they produce the same snapshot.

Definition 8 (Edit)**.** An edit may be *atomic* or *compound.*
(Atomic edit)**.** Let S be the set of all snapshots. An *atomic edit* is a 4-tuple $r = \left\langle \textit{filepath}\,,\ \textit{offset}\,, \textit{length}\,,\ \textit{text} \right\rangle$. We treat r as a function: $r\colon\ S \to S$. $r(s)$ is the same as s except that in $r(s)$, the *length* characters in s in the file *filepath* starting at position *offset* are replaced by *text*.[1]

(Compound edit)**.** Let S be the set of all snapshots. For all $n \geq 0$, a *compound edit* is a sequence of edits $e = \langle e_1, e_2, \ldots, e_n, \rangle$. We treat e as a function $e\colon\ S \to S$ such that $e(s) = e_n(e_{n-1}(\ldots(e_2(e_1(s)))))$.

For example, the atomic edit $e_1 = \left\langle \text{foo.txt}\,,\ 0\,,\ 0\,,\ \text{"public"} \right\rangle$ adds the word "public" at the beginning of `foo.txt`. After that, the atomic edit $e_2 = \left\langle \text{foo.txt}\,,\ 1\,,\ 5\,,\ \text{"rivate"} \right\rangle$ replaces "ublic" with "rivate", constructing the word "private"; and after that, the atomic edit $e_3 = \left\langle \text{foo.txt}\,,\ 0\,,\ 7\,, \text{""} \right\rangle$ deletes the word "private". An example compound edit is $\langle e_1, \langle e_2, e_3 \rangle \rangle$.

[1]Other definitions for the atomic edit are possible. For example, instead of using character offsets to indicate where to change the text, an atomic edit could specify the surrounding text. Using such a definition would change the examples in this dissertation, but not the concepts it presents.

Definition 9 (Applicability)**.** Let S be the set of all snapshots. An atomic edit $r = \langle \text{filepath}, \text{offset}, \text{length}, \text{text} \rangle$ is applicable to a snapshot $s \in S$ if the file *filepath* has at least *length* + *offset* characters. A compound edit $e = \langle e_1, e_2, \ldots, e_n \rangle$ is applicable to a snapshot $s \in S$ if $e_1, e_2, \ldots, e_n$ can be applied in sequence to s. More formally, e is applicable to s iff e_1 is applicable to s, e_2 is applicable to $e_1(s)$, …, and e_n is applicable to $e_{n-1}(e_{n-2}(\ldots(e_2(e_1(s)))))$.

If an edit e is not applicable to a snapshot s, $e(s)$ is undefined.

Definition 10 (Development history)**.** A development history is a compound edit that is applicable to the empty snapshot, $\emptyset$.

Definition 11 (Development history view)**.** Let h, h' be two development histories. Then we call h' a view of h (and h a view of h') iff $h(\emptyset) = h'(\emptyset)$.

There are three history manipulation primitives: collapse, expand, and swap. *Collapse* replaces a sequence of edits by a compound edit that consists of that sequence. *Expand* is the reverse of collapse; it replaces a compound edit by the sequence of its component parts. *Swap* swaps the location of two edits. These three primitives are sufficient to express all of Codebase Manipulation's high-level granularity transformations.

Definition 12 (Collapse)**.** For all compound edits $e = \langle \ldots, e_{i-1}, e_i, \ldots, e_j, e_{j+1}, \ldots \rangle$, $collapse(e, i-1, j-1)$ returns $\langle \ldots, e_{i-1}, \langle e_i, \ldots, e_j \rangle, e_{j+1}, \ldots \rangle$.

For example, $collapse(\langle \langle \text{foo.txt}, 0, 0, \text{“public”} \rangle, \langle \text{foo.txt}, 1, 5, \text{“rivate”} \rangle, \langle \text{foo.txt}, 0, 7, \text{“”} \rangle \rangle, 0, 1) = \langle \langle \langle \text{foo.txt}, 0, 0, \text{“public”} \rangle, \langle \text{foo.txt}, 1, 5, \text{“rivate”} \rangle \rangle, \langle \text{foo.txt}, 0, 7, \text{“”} \rangle \rangle$.

Definition 13 (Expand)**.** For all compound edits $e = \langle \ldots, e_{i-1}, \langle e_i, \ldots e_j \rangle, e_{j+1}, \ldots \rangle$, $expand(e, i-1)$ returns $\langle \ldots, e_{i-1}, e_i, \ldots, e_j, e_{j+1}, \ldots \rangle$.

Definition 14 (Swap)**.** For all development histories $h = \langle \ldots, e_i, \ldots, e_j, \ldots \rangle$, $swap(h, i-1, j-1)$ returns $\langle \ldots, e'_j, \ldots, e'_i, \ldots, \rangle$ if the resulting sequence of edits is applicable to an empty snapshot $\emptyset$; otherwise, it returns h unmodified. When the edit that comes later in the history (e_j) depends on an edit between the reordered edits ($e_i, \ldots, e_j$), the edits between the reordered edits ($e'_i, \ldots, e'_j$) might need to be modified to ensure that the resulting history reaches the same snapshot after applying the edit that comes earlier in the history (e'_i). Algorithm 1 details swap. Our algorithm uses the operational transform [142] to compute how two adjacent edits should be swapped.

Algorithm 1 Swaps the locations of two edits in the input history. Returns the resulting history if successful.

```
1: procedure SWAP($h$ : history, $i_1$ : int, $i_2$ : int)
2:     $high$ : int ← MAX($i_1, i_2$); $low$ : int ← MIN($i_1, i_2$)
3:     return MOVE(MOVE($h, high, low$), $low + 1, high$)
4: end procedure

   Moves the edit at position from to position to. Returns the resulting history if successful.
5: procedure MOVE($h$ : history, from : int, to : int)
6:     $h'$ : history ← $h$                                    ▷ Copy history.
7:     if $from = to$ then return $h'$
8:     end if
9:     $low$ : int ← MIN(from, to); $high$ : int ← MAX(from, to)
10:    for $i$ : int ← $low,\ high - 1$ do
11:        if $from > to$ then                          ▷ Edit needs to move backwards.
12:            $index$ : int ← $high + low - i$
13:        else                                         ▷ Edit needs to move forwards.
14:            $index$ : int ← $i$
15:        end if
16:        $h'$ ← SWAPADJACENT($h', index$)
17:    end for
18:    if ISHISTORY($h'$) then return $h'$
19:    else return $h$
20:    end if
21: end procedure

   Swaps $h[index]$ and $h[index + 1]$, transforming as necessary.
22: procedure SWAPADJACENT($h$ : history, $index$ : int)
23:     ▷ $e_1$ (resp. $e_2$) is the modified version of $h[index + 1]$
          (resp. $h[index]$) defined by the operational transform.
24:     $\langle e_1, e_2 \rangle$ ← OT($h[index], h[index + 1]$)
25:     return $h[0, index - 2] + [e_1, e_2] + h[index + 1, len(h) - 1]$
26: end procedure
```

Definition 15 (Granularity transformation)**.** Let H be the set of all development histories. A granularity transformation is a function $g\colon H \to H$ that takes a development history and produces another development history by applying a series of collapse, expand, and swap operations. In other words, g is a sequence of history manipulation primitives.

For simplicity of exposition, this dissertation gives algorithms for development histories with a single linear branch of development. Our work generalizes to multiple developers working concurrently and to using branches.

5.2 Codebase Manipulation: Multi-Grained Views of a History

Different development tasks require accessing the development history at different granularities. Finding the cause of a regression failure is best on a history of compilable snapshots. Studying fine-grained change patterns [116] or backtracking [154] requires the finest possible granularity. Understanding how a defect was fixed requires seeing two snapshots: one before the defect repair began and one after the repair completed. Manually examining the history may require the history as a developer created it.

In addition to different granularities, the above tasks sometimes require the development history to be restructured. For example, understanding a defect fix may require reordering the history to move unrelated changes performed during the fix, so the developer can ignore them.

Current approaches to maintaining development histories are inflexible. Manually-recorded version control histories are coarse-grained, while automatically-recorded histories that record all developer actions are extremely fine-grained but do not offer tools to change the granularity [98, 117, 153]. Codebase Manipulation mitigates the *inflexibility* of current development histories by (1) automatically recording a fine-grained development history and (2) providing the developer with tools to manipulate the granularity of the history. Using the algorithms described in this section, a developer who wishes to find the cause of a regression failure can convert an automatically-recorded history into one of only compilable edits and use history bisection (see Section 5.4.2) on that history. Then, to manually inspect the evolution of a class, the developer can convert the history into one that groups together changes based on the file it affected. Finally, to better understand the development workflow, the developer can group all contiguous edits together.

Some development tasks require the developer to view the history at *multiple* granularities. Codebase Manipulation supports these tasks by allowing the developer to repeatedly manipulate the granularity of the history, ensuring that all history manipulations are reversible.

Codebase Manipulation expresses two fundamental granularity transformations, GROUPCONSECUTIVE and GROUP, using the three history manipulation primitives defined in Section 3.1.

If the history does not require reordering, the developer can use GROUPCONSECUTIVE (Algorithm 2) to collapse consecutive edits. The developer specifies which consecutive edits should be collapsed by implementing a DECIDER routine, which returns one of three values: `new` indicates that the current edit should start a new group; `current` indicates that the current edit should be combined with the previous one; `expand` indicates that the current edit belongs to multiple

Algorithm 2 Returns a view of the input history where consecutive edits (as determined by DECIDER) are collapsed.

```
    ▷ DECIDER: $(snapshot, edit) \to \{\texttt{current}, \texttt{new}, \texttt{expand}\}$
 1: procedure GROUPCONSECUTIVE($h$ : $history$, DECIDER)
 2:     $counter \leftarrow 1$
 3:     procedure ARRANGER($s, e$)
 4:         $decision \leftarrow$ DECIDER($s, e$)
 5:         if $decision = \texttt{expand}$ then                    ▷ multi-group edit.
 6:             return ⟨"arbitrary", "non-singleton", "set"⟩
 7:         else if $decision = \texttt{new}$ then                  ▷ Create a new group.
 8:             $counter \leftarrow counter + 1$
 9:         end if
10:         return ⟨"Group" + $counter$⟩
11:     end procedure
12:     return GROUP($h$, ARRANGER)
13: end procedure
```

groups and should be expanded. GROUPCONSECUTIVE processes the input history sequentially and passes each edit and the corresponding snapshot to the DECIDER.

If the history requires reordering, the developer can use the more generic GROUP (Algorithm 3). The developer implements the logic for ARRANGER, which returns the set of group names that an edit belongs to. If the ARRANGER returns multiple group names, indicating that the edit belongs to multiple groups, GROUP attempts to expand the current edit (lines 7–12). If GROUP successfully processes a history, edits belonging to the same group are collapsed, each into its own compound edit.

GROUP and GROUPCONSECUTIVE are powerful and enable expressing interesting history transformations, including producing the following histories:

Compilable code. A compilable code history consists only of edits that produce compiling snapshots. This history view is useful for analyses that only apply to compilable code, e.g., history bisection of test failures (Section 5.4.2). GROUPCOMPILABLE (Algorithm 4) groups consecutive edits of a history into a compilable code history.

File-level change. A file-level change history groups together all changes to each file, and separates changes to different files. This history view is useful for manual inspection, and many version control systems provide diff commands that create a view similar to this history. GROUPFILES (Algorithm 5) rewrites a history into a file-level change history.

Coherent-edit change. A coherent-edit change history groups together consecutive changes

Algorithm 3 Returns a view of the input history that brings together edits for which ARRANGER returns the same value.

```
1:  procedure GROUP($h : history$, ARRANGER : $(snapshot, edit) \rightarrow Set(string)$)
2:      $intervals : \{string \rightarrow \langle int, int\rangle\} \leftarrow$ Empty map.   ▷ From group name to first and last edits in group.
3:      $h' : history \leftarrow h$; $snapshot : snapshot \leftarrow \emptyset$; $i : int \leftarrow 0$ ▷ $i$ : Index up to which $h'$ has been processed.
4:      while $i < len(h')$ do
5:          $edit : edit \leftarrow h'[i]$; $snapshot' : snapshot \leftarrow edit(snapshot)$
6:          $edit\_groups : string[\,] \leftarrow$ ARRANGER$(snapshot', edit)$   ▷ Compute the group $edit$ belongs to.
7:          if $size(edit\_groups) > 1$ then   ▷ $edit$ belongs to multiple groups, try expanding it.
8:              $h'' \leftarrow$ EXPAND$(h', i)$
9:              if $h'' = h'$ then Print "History cannot be grouped."; return $h$   ▷ Expand failed, return $h$.
10:             end if
11:             $h' \leftarrow h''$; continue
12:         end if
13:         $\langle h', intervals\rangle =$ MOVETOGROUP$(h', intervals, i, edit\_groups[0])$
14:         $snapshot \leftarrow snapshot'$; $i \leftarrow i + 1$
15:     end while
16:     for all $\langle gname, interval\rangle \in intervals$ do   ▷ Collapse each interval.
17:         $\langle s, e\rangle \leftarrow interval$; $h' \leftarrow$ COLLAPSE$(h', s, e)$
18:     end for
19:     return $h'$
20: end procedure
    Moves the edit at $i$ to the end of the given edit group.
21: procedure MOVETOGROUP($h : history, intervals, i : int, gname$)
22:     $interval : \langle int, int\rangle \leftarrow intervals[gname]$
23:     if $interval =$ null then $intervals[gname] \leftarrow \langle i, i\rangle$   ▷ Edit creates a new group.
24:     else   ▷ Move the edit to the end of an existing interval.
25:         $\langle s, e\rangle \leftarrow interval$; $intervals[gname] \leftarrow \langle s, e + 1\rangle$
26:         if $i \neq e + 1$ then   ▷ Move the edit.
27:             $h' \leftarrow$ MOVE$(h, i, e + 1)$
28:             if $h = h'$ then Print "History cannot be grouped."; return $h$   ▷ Move failed, return $h$.
29:             end if
30:             for all $\langle gname, interval'\rangle \in intervals$ do   ▷ Shift all intervals that are later in the history.
31:                 $\langle s', e'\rangle \leftarrow interval'$
32:                 if $s' \geq e + 1$ then $intervals[gname] \leftarrow \langle s' + 1, e' + 1\rangle$   ▷ Shift the interval.
33:                 end if
34:             end for
35:         end if
36:     end if
37:     return $\langle h', intervals\rangle$
38: end procedure
```

Algorithm 4 GroupCompilable: Returns a view of the input history such that the snapshot produced by each edit compiles.

```
    ▷ COMPILE: $snapshot \to bool$
1: procedure GROUPCOMPILABLE($h : history$, COMPILE)
2:     procedure DECIDER($s : snapshot, e : edit$)
3:         if COMPILE($s$) then return new
4:         else return current
5:         end if
6:     end procedure
7:     return GROUPCONSECUTIVE($h$, DECIDER)
8: end procedure
```

Algorithm 5 GroupFiles: Returns a view of the input history grouping all of each file's edits together.

```
 1: procedure GROUPFILES($h : history$)
 2:     procedure ARRANGER($s : snapshot, e : edit$)
 3:         return EDITTOFILENAMES($e$)
 4:     end procedure
 5:     return GROUP($h$, ARRANGER)
 6: end procedure
    Recursively searches the input edit and returns the file names of all files that are affected by this edit.
 7: procedure EDITTOFILENAMES($e : edit$)
 8:     if $e$ is an atomic edit then $\langle file\_path, \_, \_, \_\rangle \leftarrow e$; return $\{file\_path\}$
 9:     end if
10:     $result : Set(string) \leftarrow \emptyset$
11:     for all $edit \in e$ do                              ▷ $e$ is a compound edit.
12:         $result \leftarrow result \cup$ EDITTOFILENAMES($edit$)
13:     end for
14:     return $result$
15: end procedure
```

that occur together spatially in the source code. GROUPCOHERENT (Algorithm 6) rewrites a history into a coherent-edit change history by detecting contiguous edits. This history view is useful for manual inspection and for certain backtracking and workflow analyses.

5.3 Bread: Codebase Manipulation for Eclipse

We have implemented Bread, a Codebase Manipulation prototype for the Eclipse IDE. Bread automatically records a fine-grained development history and enables the developer to access multi-

Algorithm 6 GroupCoherent: Returns a view of the input history grouping all contiguous consecutive edits together.

```
1:  procedure GROUPCOHERENT($h$ : $history$)
2:      $counter : int \leftarrow 1$
3:      $regions : \{string \rightarrow \langle string, int, int \rangle\} \leftarrow$ Empty map.  ▷ Maps group name to file region; see line 33
4:      procedure ARRANGER($s$ : $snapshot$, $e$ : $edit$)
5:          $\langle edit\_groups, regions', counter' \rangle \leftarrow$ PROCESSEDIT($e, regions, counter$)   ▷ See line 34
6:          if $len(edit\_groups) = 1$ then   ▷ $e$ belongs to a unique group. Update states.
7:              $regions \leftarrow regions'$; $counter \leftarrow counter'$
8:          end if
9:          return $edit\_groups$
10:     end procedure
11:     $h' \leftarrow h$   ▷ Group $h'$ until all intervals are separated.
12:     while true do
13:         $h'' \leftarrow$ GROUP($h'$, ARRANGER)
14:         if $h'' = h'$ then return $h$   ▷ GROUP failed. Return $h$
15:         end if
16:         $h' \leftarrow h''$
17:         if REGIONCOMBINES($regions$) then   ▷ Reset.
18:             $counter \leftarrow 1$; $regions \leftarrow$ Empty map.
19:         else return $h'$
20:         end if
21:     end while
22: end procedure
23: Returns true if any two regions in the input map intersect with each other.
24: ▷ $region : \langle string, int, int \rangle$
25: procedure REGIONCOMBINES($regions : string \rightarrow region$)
26:     return $\{\exists \langle \_, reg_1 \rangle, \langle \_, reg_2 \rangle \in regions \mid reg_1 \neq reg_2 \wedge$ COMBINES$(reg_1, reg_2)\}$
27: end procedure
28: Returns true if the input intervals intersect with each other.
29: procedure COMBINES($i_1 : \langle string, int, int \rangle, i_2 : \langle string, int, int \rangle$)
30:     $\langle f_1 : string, s_1 : int, e_1 : int \rangle \leftarrow i_1$; $\langle f_2 : string, s_2 : int, e_2 : int \rangle \leftarrow i_2$
31:     return $(f_1 = f_2)$ and $((s_1 \geq s_2 - 1$ and $s_1 \leq e_2 - 1)$ or $(e_1 \geq s_2 - 1$ and $e_1 \leq e_2 - 1))$
32: end procedure
```

grained views of that history. Bread removes the burden of manual development history creation, improves existing historical analyses, and simplifies the implementation of new historical analyses. Bread satisfies the following requirements:

Complete history: Bread should record every developer action (including ones that the developer undoes) and the resultant code changes.

Algorithm 6 (continued) ProcessEdit: computes which groups the input edit *e* belongs to. Returns a triple: (1) the group names that *e* belongs to, (2) the next state of *regions*, and (3) the next value of the counter *c*.

```
33: ▷ $regions : string \rightarrow \langle string, int, int\rangle$ maps from group name to file region. A file region ⟨file, start index,
    end index⟩ is a group of contiguous consecutive edits.
34: procedure ProcessEdit($e : edit, regions, c : int$)
35:     $result : Set(string) \leftarrow \emptyset$; $regions' \leftarrow regions$                    ▷ Copy regions.
36:     if $e$ is an atomic edit then
37:         $\langle file\_path,\ offset,\ length,\ text\rangle \leftarrow e$
38:         $elength \leftarrow len(text) - length$; $eend \leftarrow offset + elength$
39:         for all $\langle gname, region\rangle \in regions'$ do
40:             $\langle fpath, start, end\rangle \leftarrow region$
41:             if Combines($\langle file\_path, offset, eend\rangle, region$) then        ▷ Edit combines with the region.
42:                 $end' \leftarrow end + elength$; $result \leftarrow result \cup \{gname\}$
43:                 $regions'[gname] \leftarrow \langle fpath, start, end'\rangle$
44:             else if $file\_path = fpath$ and $eend < start$ then        ▷ Move region forward or backward.
45:                 $start' \leftarrow start + elength$; $end' \leftarrow end + elength$
46:                 $regions'[gname] \leftarrow \langle fpath, start', end'\rangle$
47:             end if
48:         end for
49:         if $len(result) = 0$ then                                   ▷ e belongs to a new group.
50:             $gname \leftarrow$ "Group" $+ c$; $c \leftarrow c + 1$
51:             $regions'[gname] \leftarrow \langle file\_path, offset, eend\rangle$; $result \leftarrow result \cup \{gname\}$
52:         end if
53:         return $\langle result, regions', c\rangle$
54:     end if
55:     ▷ e is a sequence of edits
56:     for all $edit \in e$ do
57:         $\langle edit\_gnames, regions'', c\rangle \leftarrow$ ProcessEdit($edit, regions', c$)
58:         $result \leftarrow result \cup edit\_gnames$; $regions' \leftarrow regions''$
59:         if $len(result) > 1$ then                 ▷ Edit belongs to multiple groups. No need to continue.
60:             return $\langle result, regions', c\rangle$
61:         end if
62:     end for
63:     return $\langle result, regions', c\rangle$
64: end procedure
```

Easy-to-use history: Bread's multi-grained history views should be easy to use by the developer, and by automated analysis tools.

Unobtrusive recording: Bread should not interfere with existing development tools. It should neither slow down the developer's IDE nor affect manually-managed version control histories.

5.3.1 Bread design and implementation

Bread is an open-source Eclipse plug-in, publicly available at: `https://bitbucket.org/kivancmuslu/chronos`. Bread automatically records the fine-grained history into a Git repository. Each developer action, even one that does not alter the source code, results in a commit, with the log message storing information on the action itself. Bread is built on top of Solstice [111] (Section 3.4), an Eclipse plug-in for Codebase Replication [105, 106] (Chapter 2) that facilitates IDE interactions (Figure 5.1). Solstice maintains a copy of the developer's code in parallel to the developer's work, detects all code changes, and provides Bread with observer patterns for the changes.

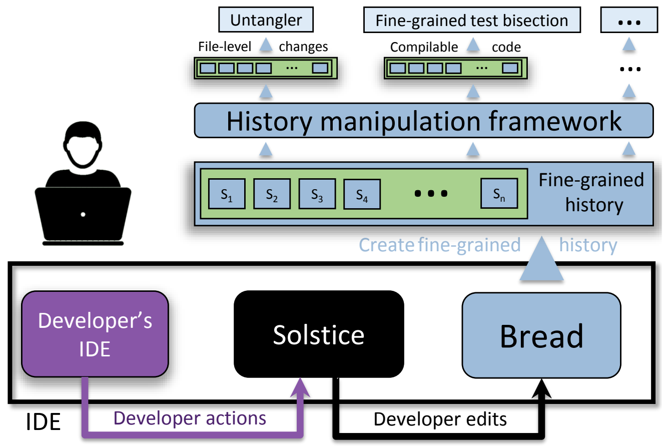


Figure 5.1: Bread architecture. Bread (blue) extends Solstice (black) to automatically maintain the fine-grained development history. Bread's history manipulation framework offers multi-grained views of the fine-grained history by rewriting the history into coarser granularities. The developer can inspect — manually or through an automated analysis tool — the view that is the most suitable for the underlying task.

Bread satisfies the complete-history requirement by detecting every developer action within Eclipse via the Eclipse's API, and recording all such actions and every textual change to the source code.

Bread satisfies the easy-to-use requirement by providing a history manipulation framework to automatically transform the recorded development history into coarser granularities. The converted histories are themselves Git repositories, which can be inspected manually and interface with automated tools. (Section 5.4 evaluates how well Codebase Manipulation and Bread satisfy this requirement.)

Bread satisfies the unobtrusive-recording requirement by storing its fine-grained Git repository in a unique folder on the filesystem. The developer may continue to use any version control system, including Git, to create a manual history in parallel, and tools can access both the codebase and the manual history. Git is fast enough that Bread's overhead is negligible. (Section 5.4 evaluates how well Bread satisfies this requirement.)

5.3.2 Bread Limitations

Bread is susceptible to Solstice's design limitations. Solstice detects source code changes through the IDE API; if the source code is changed outside the IDE, Bread will not record these changes immediately. Developers rarely edit outside of their preferred IDE, but to mitigate this limitation, each time the IDE is opened, Bread checks for any changes to the source code that may have taken place and creates an edit containing these external changes. Bread could have avoided this limitation by using OS-level, file-system listeners to detect changes to the source code. However, this approach would prevent Bread from detecting changes that are not written to the file system, such as unsaved changes in editor buffers. Additionally, Solstice detects some developer actions initiated via tools as typing actions, and therefore Bread records them as such. For example, Bread records Eclipse refactorings as a series of text replace operations to the source code. Thus, Bread is complete in its recording, but inherits Solstice's limitations in recognizing how some actions are initiated. Improvements to Solstice would be immediately reflected in Bread.

5.4 Expressiveness Evaluation

We evaluated Bread's usability in terms of the expressiveness of the histories it can create. We identified three common development use cases that benefit from Codebase Manipulation's multi-grained views. First, developers often perform multiple tasks on a project concurrently, such as fixing bugs, adding features, refactoring, and improving documentation. While the history reflects these tasks being performed concurrently, developers may wish to untangle the changes relevant to each task. This, for example, simplifies later undoing one of the tasks without affecting the other tasks. Bread's multi-grain history supports this use case, and we have built Untangler, a tool that uses Bread to help developers untangle changes (Section 5.4.1). Second, developers often use program analyses that rely on development histories. The granularity of the history may affect the analysis, and Bread's granularity manipulation capabilities support this use case. We have built fine-grained test bisection, a tool that demonstrates this feature (Section 5.4.2). Third, developers often manually examine a project's history to understand the project's evolution. Bread supports this use case by enabling the developer to change the history's granularity to improve evolution understanding (Section 5.4.3).

5.4.1 Untangling Code Changes

Rewriting development histories to improve usability can be helpful for maintenance tasks. Developers often tangle multiple tasks during development. For example, while fixing a defect, a developer may observe a need for a refactoring, a new feature, or improved documentation; the developer may implement those changes in the middle of the defect fix. This causes the manually-created version control history to tangle changes relevant to multiple tasks, which later makes it difficult to understand the changes individually, such as to undo one of them.

Some VCSs provide a staging area and rebase commands to simplify untangling changes at the time they are created or before they are shared with others. However, these features cannot be used after the fact and do not address interactions between the changes, e.g., if adding a feature adds a line of code that fixing a defect removes. Heuristics can help guide untangling large edits but also fail to capture interactions between the changes [72, 92].

To address this problem, we have built Untangler, a new interactive analysis tool that helps developers reorganize and rewrite a history into one that brings together changes relevant to particular development tasks. Untangler can work with fine-grained development histories, such as the ones automatically recorded by Bread, or with manually-created version control histories that Untangler decomposes into a fine-grained history. Untangler is open-source: `https://bitbucket.org/LukeSwart/untangler`.

Unlike prior work, Untangler (Algorithm 7) does not rely on heuristics on top of joined diffs of a set of changes. Instead, it is an interactive history rewriting process that uses a history.

Untangler uses Codebase Manipulation to first rewrite the history into a coherent-edit change history and then into a file-level change history. Then, Untangler creates an empty *target* history in which the untangled edits will be stored. Finally, Untangler interactively allows the developer to expand and collapse edits. Once a developer is satisfied with a set of edits, *committing* those edits adds them to the target history and removes them from the editable view.

Case study: To explore the capabilities of Untangler, a developer (who is the author of this dissertation) used Bread to record a history while building a Java graph library via test-driven development. The developer also maintained a manual development history, aiming to create a snapshot whenever a high-level task was complete. The developer started from skeleton code with a Javadoc specification and 95 failing tests. At the end of the implementation, the manual development history had 6 snapshots, and the Bread history had 7,132 snapshots.

The library requires directed and undirected edges and graphs. In the manual history, the developer implemented both directed and undirected edges within the first edit. In the second edit, the developer realized that these implementations shared significant common code and refactored the edge implementation into an abstract class with the common functionality. In the third edit, the developer implemented both graph implementations, and then again, in the fourth edit, refactored

Algorithm 7 Untangler first computes the coherent-edit and file-level-change views of the fine-grained history and then allows the developer to expand, collapse, and commit changes to create the history the developer wants.

```
 1: procedure UNTANGLER($h : history$)
 2:     $h\_coherent : history \leftarrow$ GROUPCOHERENT($h$); $h\_file : history \leftarrow$ GROUPFILES($h\_coherent$)
 3:     $h' : history \leftarrow h\_file$                                ▷ Copy file-level-change view.
 4:     repeat
 5:         ▷ Visualize edits in $h'$. The developer makes a choice ($c$ : {Exit, Expand, Collapse, Commit})
              and selects a list of edits associated with this choice. $idxes$ are edit indices, in sorted order.
 6:         $\langle c, idxes \rangle \leftarrow$ INTERACT($h'$)
 7:         if $c =$ Expand then                                     ▷ Expand each of the selected edits.
 8:             $offset : int \leftarrow 0$
 9:             for all $index : int \in idxes$ do
10:                 $shifted\_index : int \leftarrow index + offset$; $offset \leftarrow offset + len(h'[shifted\_index]) - 1$
11:                 $h' \leftarrow$ EXPAND($h', shifted\_index$)
12:             end for
13:         else if $c =$ Collapse then                    ▷ Make the edits contiguous in the history.
14:             $h'' \leftarrow$ MOVEALL($h', idxes[1, len(idxes) - 1], idxes[0]$)
15:             if $h' = h''$ then Display "Cannot collapse selected edits."               ▷ Move failed.
16:             else $h' \leftarrow h''$; $h' \leftarrow$ COLLAPSE($h', idxes[0], idxes[0] + len(idxes) - 1$)
17:             end if
18:         else if $c =$ Commit then                                       ▷ Move edits to beginning
19:             $h'' \leftarrow$ MOVEALL($h', idxes, 0$)
20:             if $h' = h''$ then Display "Cannot commit selected edits."                 ▷ Move failed.
21:             else
22:                 $h' \leftarrow h''$; $s : snapshot \leftarrow$ GETSNAPSHOT($h', len(idxes)$)
23:                 Sync and commit $s$. Remove edits at $idxes$ from the view.
24:             end if
25:         end if
26:     until $c =$ Exit
27: end procedure
28: procedure MOVEALL($h : history, idxes : int[], after : int$)   ▷ Moves edits at $idxes$ after location $after$.
29:     $h' \leftarrow h$                                                          ▷ Copy history.
30:     for $i \leftarrow 0, len(idxes) - 1$ do
31:         $h'' \leftarrow$ MOVE($h', idxes[i], after + i$)
32:         if $h' = h''$ then return $h$
33:         end if
34:         $h' \leftarrow h''$
35:     end for
36:     return $h'$
37: end procedure
```

this duplication to group the common functionality. The fifth edit removed some unnecessary code left over from the second edit. These five edits represent six high-level software engineering tasks: implementing (1) the abstract edge class, (2) the undirected edge, (3) the directed edge, (4) the abstract graph class, (5) the undirected graph representation, and (6) the directed graph representation. Using today's state-of-the-art VCSs and research techniques, it is difficult to untangle the six high-level tasks in the history.

Running Untangler on this history automatically proposed exactly the six high-level software development tasks because each of them was a coherent edit. In this case, all the developer had to do was commit each of the edits suggested by Untangler to produce an untangled history. In other cases, the developer may need to expand and collapse edits, implicitly reordering them when collapsing and committing. This anecdotal evidence demonstrates that Untangler can successfully untangle development histories.

5.4.2 Fine-Grained Test Bisection

Automated history analysis can help some development tasks. For example, history bisection of test failures (also known as test bisection) [61], identifies which edit introduced a regression failure. It uses binary search and a test case on a VCS repository to find the first snapshot such that the test fails. (More generally, history bisection can apply to any property of a snapshot, beyond just test results.) However, the effectiveness of test bisection depends on the quality and granularity of the history. Given a fine-grained history, test bisection returns a concise description of the failure-introducing change. However, given a coarse, manually-created history, test bisection can only identify a large change as responsible for introducing the test failure. Manually-created development histories are unlikely to be optimal for test bisection. Existing approaches, such as delta debugging [156, 157] improve and simplify test bisection results but can produce partial edits that divide atomic developer actions. For example, these approaches may find that while a refactoring does not break a test, half of that refactoring does, and this may mask another, later cause of the failure.

To address this problem and to demonstrate another use of Bread, we built Bisector, a fine-grained test bisection tool that uses the fine-grained development history to pinpoint real developer actions that cause regression failures. Bisector is available as an open-source tool: `https://bitbucket.org/kivancmuslu/chronos-test-bisection`. Fine-grained test bisection (Algorithm 8) uses the rich information inside the fine-grained development history to compute the minimal cause of the regression failure, exactly as it is introduced during development. Fine-grained test bisection guarantees to identify the most recent cause of the failure by bisecting the more-recent half of the history first, regardless of the bisection results on the less-recent half and how many times the test regresses during the fine-grained history. The latest regression failure is probably on code that is most familiar to the developer. Building Bisector was easy: it took one developer three days and required only 670 lines of Java.

Algorithm 8 Fine-grained test bisection (FGBISECT) improves test bisection by pinpointing the cause of a regression failure exactly as it was introduced during development.

```
 1: procedure FGBISECT($h$ : $fine\_grained\_history$, TESTRUNNER: $\{snapshot \to bool\}$)
 2:     $h'$ : $history \leftarrow$ GROUPCOMPILABLE($h$)
 3:     return BISECT($h'$, TESTRUNNER, $0$, $len(h') - 1$)
 4: end procedure
 5: ▷ $left$ : earliest known failure for the latest regression, $right$ : latest known success.
 6: procedure BISECT($h$ : $history$, TESTRUNNER : $\{snapshot \to bool\}$, $left$ : $int$, $right$ : $int$)
 7:     if $left = right - 1$ then                                          ▷ Base case.
 8:         $left\_snap$ : $snapshot \leftarrow$ GETSNAPSHOT($h, left$)
 9:         $left\_result$ : $bool \leftarrow$ TESTRUNNER($left\_snap$)
10:         $right\_snap$ : $snapshot \leftarrow$ GETSNAPSHOT($h, right$)
11:         $right\_result$ : $bool \leftarrow$ TESTRUNNER($right\_snap$)
12:         if $left\_result$ and not($right\_result$) then
13:             return $h[right]$
14:         end if
15:         Print "Cannot bisect the test"
16:         return null
17:     end if
18:     ▷ Recursive case.
19:     $mid \leftarrow (right + left)/2$                        ▷ Integer division, round down.
20:     $mid\_snap$ : $snapshot \leftarrow$ GETSNAPSHOT($h, mid$)
21:     $mid\_result$ : $bool \leftarrow$ TESTRUNNER($mid\_snap$)
22:     ▷ Bisect the upper part of the history regardless of the current
          result to find the latest regression.
23:     $upper\_result$ : $edit \leftarrow$ BISECT(h, TESTRUNNER, $mid$, $right$)
24:     if ($mid\_result$ = false) and ($upper\_result$ = null) then
25:         return BISECT($h$, TESTRUNNER, $left$, $mid$)
26:     end if
27:     return $mid\_result$
28: end procedure
```

We used Bisector on the fine-grained history recorded while developing the Java graph library from Section 5.4.1. Running test bisection — for each test — on the manual development history yielded no results. However, running Bisector on the Bread history yielded a bisectable test. The test failure happened while the developer was in the middle of implementing the condition of an `if` clause. The correct implementation has a negated condition, but the developer first implemented the condition without the negation, causing a test failure. This example illustrates Bisector's potential to pinpoint test failures that developers introduce unintentionally.

5.4.3 Understanding Software Evolution

Understanding software evolution is an important and challenging software engineering task [8]. For example, understanding how a class evolves over time can help identify bugs [91]. But today, the state-of-the-practice of understanding code element evolution is ad-hoc. Developers can use version control to look at individual snapshots or diffs, use impact analysis to identify potential consequences of an edit [4], and examine thin program slices to understand what parts of a program affect an element [141]. However, these approaches focus on individual snapshots or pairs of snapshots and fail to capture long-term *evolution*.

To address this challenge, we have designed the CodeEvolve algorithm (Algorithm 9). Using Codebase Manipulation, CodeEvolve transforms the history into the compilable-code granularity and then further transforms the history to contain only the edits relevant to the element of interest.[2]

5.5 Performance Evaluation

To evaluate if Bread is unobtrusive, we empirically measured Bread's performance. We ran all experiments on a modern laptop (Intel Core i7-4650U CPU, 8GB ram, SSD hard drive). We focus on scalability, overhead on developer actions, and recording delay.

Scalability. Since code changes are as frequent as individual keystrokes, Bread must scale to handle large histories.

Bread stores each code change as a Git changeset. To understand how well Git scales with development history size, we designed an experiment that creates a random fine-grained development history by invoking `git commit` after each single-character edit. The experiment creates 50 random files of 100–200 lines with each line of 40–80 characters. The experiment measures the time it takes to execute `git commit` after each edit. Just like Bread, the experiment calls garbage collection (`git gc`) as soon as the number of loose Git objects reaches 300. The garbage collection

[2]Identification of these edits requires AST differencing (ASTDIFF in Algorithm 9), such as the Diff/TS algorithm [70].

Algorithm 9 CodeEvolve: Returns a finest-grain view of the input history such that the snapshot produced by each edit modifies at least one of the code elements of interest.

```
1: procedure $\textsc{CodeEvolve}(h : \mathit{fine\_grained\_history}, \textsc{IncludeNode} : \mathit{ASTNode} \rightarrow \mathit{bool}, \textsc{Compiler} : \mathit{snapshot} \rightarrow \mathit{AST})$
2:     $\mathit{prev\_ast} : \mathit{AST} \leftarrow$ empty AST
3:     procedure $\textsc{Decider}(s : \mathit{snapshot}, e : \mathit{edit})$
4:         $\mathit{ast} : \mathit{AST} \leftarrow \textsc{Compiler}(s)$
5:         ▷ Diff the current and the previous AST.
6:         $\langle \mathit{added}, \mathit{modified}, \mathit{removed} : \mathit{ASTNode}[] \rangle \leftarrow \textsc{ASTDiff}(\mathit{prev\_ast}, \mathit{ast})$
7:         $\mathit{prev\_ast} \leftarrow \mathit{ast}$
8:         ▷ Return true if the AST diff wrt the previous AST contains a node of interest.
9:         $\mathit{affected} \leftarrow \mathit{added} \cup \mathit{modified} \cup \mathit{removed}$; return $\{\exists\, n \in \mathit{affected} \mid \textsc{IncludeNode}(n)\}$
10:    end procedure
11:    $h \leftarrow \textsc{GroupCompilable}(h)$; $h \leftarrow \textsc{GroupConsecutive}(h, \textsc{Decider})$
12:    ▷ Compute frames (snapshots) of interest.
13:    $\mathit{current\_frame} : \mathit{snapshot} \leftarrow \emptyset$; $\mathit{frames} : \mathit{snapshot}[] \leftarrow$ Empty array.
14:    for all $e : \mathit{edit} \in h'$ do
15:        $\mathit{current\_frame} \leftarrow e(\mathit{current\_frame})$; $\mathit{frames} \leftarrow \mathit{frames} + [\mathit{current\_frame}]$
16:    end for
17:    Visualize $\mathit{frames}$.
18: end procedure
```

duration is excluded from the results because Git supports garbage collection in parallel with other Git operations, and Bread uses idle CPU cycles for garbage collection.

The average time overhead of recording edits was 37% over 367K edits. Even after 350K edits, Git takes less than 15 ms to record each edit. We conclude that Git scales well with respect to development workloads. The 37% overhead is a worst case scenario of continuously typing of the 367K characters. Future work can further improve scalability by splitting the history into multiple internal repositories.

Having found that Git scales reasonably well, we next explored Bread's overhead on developer actions.

Developer Action Overhead. As Bread tracks all developer changes at the buffer level, it listens to each keystroke. To be unobtrusive, the overhead experienced by the developer must be close to zero so that Bread does not adversely affect the developer. Figure 5.2 shows, for the most common developer actions, the IDE overhead that Bread introduces when the action is initiated programmatically. One of the most common developer actions is editing a file with keystrokes. Figure 5.2 represents such text edits as size 1. IDEs also support complex operations, such as

Operation Name	Operation Size	Initial File Size (chars)	IDE Overhead (ms)	Recording Delay (ms)	Operation Name	Operation Size	Initial File Size (chars)	IDE Overhead (ms)	Recording Delay (ms)
Text Insert	1	0	2.0	4.0	Text Delete	1	1	0.9	4.4
		100	2.1	9.5			101	2.0	7.6
		1,000	2.0	10.5			1,001	2.0	7.4
		10,000	4.3	9.6			10,001	3.6	8.5
	100	0	2.1	9.9		100	1	1.1	7.5
		100	1.9	10.1			101	1.9	7.6
		1,000	2.0	10.5			1,001	1.9	7.4
		10,000	3.6	9.6			10,001	3.6	7.8
Text Edit	1	100	1.8	10.0	File Add	1	1,000	1.6	13.2
		1,000	2.0	10.1		100		268.0	1,363.0
		10,000	3.5	10.8		1,000		5,819.2	46,008.5
	100	100	1.8	10.8	File Remove	1	1,000	1.6	9.3
		1,000	1.9	10.6		100		202.3	850.0
		10,000	3.6	10.9		1,000		4,138.4	16,801.5
Text Edit Summary			< 4.5	< 11.0	**File Edit Summary**			grows linearly with size	

Figure 5.2: Bread's overhead for developer edits. "IDE Overhead" measures the overhead imposed on the responsiveness of the IDE, and "Recording Delay" measures the delay before the fine-grained development history is up-to-date. Text operations are means over 20,000 executions. File operations are means over 200 executions.

refactoring, auto-complete, copy-and-paste, etc., which are represented as edits of size 100. The results show that the overhead is independent of the edit size, and even for large files (10,000 characters), the overhead is no more than 4.5 ms.

Manually adding and removing files in an Eclipse project are represented as 1-, 100-, and 1,000-sized file operations. File operations of size 1 are manual file generation, copy, and removal, and file operations of size 100 and 1,000 represent copying, removing, or importing a directory or an entire Eclipse project. The results suggest that the overhead for file operations increases linearly with operation size, as expected. Removals are faster than additions, and even for large operations, the overhead never exceeds a few seconds. Since Eclipse already takes several seconds to import a project with 1,000 files, the results suggest that Bread introduces negligible IDE overhead.

Recording Delay. To keep the automatically-recorded history up-to-date, the delay between when the developer makes a code change and when Bread records it should be negligible. An excellent typist types 80 words per minute, which implies a 150-ms delay between consecutive keystrokes. Software development is slower than transcribing text, so a delay to record code changes of less than 150 ms should satisfy the unobtrusive recording requirement.

Figure 5.2 shows the delay Bread incurs while recording code changes for the most common developer operations. Except for importing and deleting large Eclipse projects, recording delay is less than 11 ms. Since importing and deleting large Eclipse projects is fairly rare, and these operations already take several seconds for Eclipse to execute, we conclude that Bread records the history with negligible delay.

5.6 Contributions

Development histories are necessary for software engineering tasks, but their inflexible granularity hinders their utility. This chapter introduced multi-grained views that lets the developer access the history in the most optimal granularity for the current task. The granularity of these views are not known in advance, which necessitates the underlying technique to create and maintain a fine-grained development history. Codebase Replication (Chapter 2) trivializes how Codebase Manipulation records this fine-grained history by detecting all developer edits and applying these edits to the copy codebase, which provides access to the corresponding development snapshots. On top of this fine-grained history, Codebase Manipulation applies high-level history transformations to rewrite the history's granularity to make it more suitable for specific tasks. We have demonstrated that Codebase Manipulation is highly expressive by designing three tools that use Codebase Manipulation to aid untangling concurrent code changes, localize causes of regression failures, and understanding code evolution. Bread, an implementation of Codebase Manipulation, demonstrates that our approach is highly efficient and has negligible overhead. We publicly released all our source code. Overall, these results show promise that Codebase Manipulation can improve information retrieval tools that use the development history and can make histories more usable for manual inspection.

Chapter 6
RELATED WORK

This chapter puts the dissertation in the context of related research by discussing continuous analyses and continuous analyses frameworks (Section 6.1), speculative analyses (Section 6.2) fine-grained development history frameworks and tools built on these frameworks (Section 6.3), and use of replication to achieve different goals (Section 6.4).

6.1 Continuous Analyses

This section places Codebase Analysis in the context of related research. Section 6.1.1 discusses other approaches to building continuous analysis tools and Section 6.1.2 discusses existing continuous analysis tools and their benefits.

6.1.1 Building Continuous Analysis Tools

As we have described, an ε-continuous analysis exhibits both currency and isolation. Codebase Analysis simplifies building such analyses. Alternatively, developers can build such tools by using IDEs' APIs to listen to source code edits. For example, Eclipse's `IResourceChangeListener` [41] and `IDocumentListener` [40] APIs broadcast file-level and memory-level changes, respectively. Eclipse's Java incremental compiler [46] and reconciler compiler [45] use these APIs; however, these analyses are written by the IDE developers, and building ε-continuous analyses using these primitive APIs is prohibitively difficult for third-party developers.

Some specialized development domains make building a limited set of continuous analyses simple. For example, a spreadsheet can be thought of as an IDE for data-intensive programs that reruns these programs on every code or data update. VisiProg [71, 87] proposed to extend this paradigm to general programming languages, but Codebase Analysis is the first implementation that provides isolation and currency. As another example, live programming [17, 149, 23], which eases development by executing a fast-running program on a specific input as that program is being developed, is a special case of continuous analysis.

The rest of this section discusses alternate ways of creating continuous analysis tools and compares them to Codebase Analysis. None of the existing approaches provides both isolation and currency, although some provide one or the other.

Methods that Yield Limited Currency but Lack Isolation

IDEs provide higher-level frameworks than the primitive listeners described earlier. For example, to simplify implementing build-triggered continuous analyses, Eclipse provides Incremental Project Builders [42]. This mechanism can be used to execute an analysis on the on-disk version of the program every time the incremental compiler runs. (Note that when auto-build is enabled in Eclipse, the code builds every time it is saved to disk, so build-trigger becomes equivalent to file-change-trigger.) This mechanism enables building analyses that have some currency, although Codebase Analysis' memory-change access provides better currency by enabling the analyses to run on a more recent version of the program than one that has been saved to disk. Further, unlike Codebase Analysis, this mechanism does not allow for analysis isolation. The analyses run on the developer's on-disk copy, meaning that an impure analysis's changes directly alter the developer's code, and a developer's concurrent changes may affect the analysis.

IDEs also provide frameworks that simplify building a limited set of continuous analyses with memory-change currency. For example, Eclipse's Xtext [152] simplifies extending Eclipse to handle new languages. Xtext provides parsing, compilation, auto-complete, quick fix, and refactoring support, but is limited to building language extensions. Meanwhile Codebase Analysis provides memory-change currency for arbitrary source or binary code analyses. Similarly to Incremental Project Builders, and unlike Codebase Analysis, Xtext does not allow for analysis isolation as a developer's concurrent changes may affect the analysis. Further, Xtext does not support impure analyses.

Methods that Yield Limited Isolation but Lack Currency

Integration servers, such as Jenkins [82], can enable certain kinds of continuous analyses. An integration server maintains an isolated copy of the program under development and automatically fetches new changes, builds the program, runs static and dynamic analyses, and generates summaries for developers and project managers. However, integration servers lack currency, as they cannot be memory-change- or file-change-triggered; typically they are triggered periodically or by events such as a commit. Modern collaboration portals, such as `github.com`, `bitbucket.org`, and `googlecode.com`, integrate awareness analyses and create interfaces for developers to get feedback on the state of their programs. This is also a step toward making analyses continuous, as the portals can automate the running of analyses and can analyze multiple developers' codebases and notify developers of analysis results. However, this mechanism also lacks currency as the analyses cannot be triggered by memory changes, file changes, or even most version control operations.

IDEs provide APIs that serialize accesses to the codebase, which can ensure partial isolation. For example, Eclipse provides a Jobs API [48] that enables third-party developers to schedule jobs that access the codebase. There is no isolation: these jobs either block each other and the developer edits, or they occur concurrently on the same codebase. In contrast, Codebase Analysis can run an analysis on the copy codebase while letting the developer work, achieving true isolation, and providing native support for impure continuous analyses.

6.1.2 Existing Continuous Analysis Tools

Continuous analysis tools help developers by reducing the notification delay of code changes' effects on analysis results. For example, continuous testing [128, 129, 130] executes a program's test suite as the program is being developed. In a study, continuous testing made developers three times as likely to finish programming tasks by a deadline [129] and reduced the time needed to finish a task by 10–15% [128]. Similarly, continuous data testing greatly reduced data entry errors [110], and continuous compilation made developers twice as likely to finish programming tasks by a deadline [129]. Some continuous analyses [21, 22, 69, 109] can be speculative [19] by predicting developers' likely future actions and executing them in the background to inform developers' decision making. Such tools have the potential to further increase the benefits of continuous analyses.

Figure 6.1 lists previous continuous analysis tools of which we are aware. Although IDEs provide frameworks and APIs to simplify the creation of continuous analyses, Figure 6.1 shows that most existing third-party IDE-integrated continuous analysis tools are not ε-continuous, lacking either in isolation or currency. From the sixteen file-change-triggered and build-triggered tools in Figure 6.1, we selected the seven with evidence of development or maintenance within the last year and contacted their developers to ask if they had considered making their analyses run whenever the in-memory code changes or compiles. We received responses from the developers of four of the seven tools, GoClipse, InPlace Activator, TSLint, and TypeScript (TSLint and TypeScript are developed by an overlapping set of developers). All the developers thought making analyses continuous was a good idea, with one remarking that this would be hard to do, another that he didn't have enough time to implement this feature, and the third pointing out that part of the tool already has this continuous behavior, although not all of the tool's analyses are continuous. We

	Currency			Isolation
Tool	Trigger	Interruption	Staleness	
Quick Fix Scout [109] consequences of Eclipse quick fixes	Memory-change*	Immediately interrupt	Immediately remove	Full
Eclipse reconciler compiler [45]	Memory-change*	Immediately interrupt	Immediately remove	Developer
JKind [83] Eclipse plug-in for JKind language	Memory-change	Immediately interrupt	Immediately remove	Developer
eVHDL [54] Eclipse plug-in for VHDL language	Memory-change	Immediately interrupt	Immediately remove	Developer
Scribble [135] Eclipse plug-in for Scribble language	Memory-change	Immediately interrupt	Immediately remove	Developer
wNesC [150] Eclipse plug-in for NesC language	Memory-change	Immediately interrupt	Immediately remove	Developer
OcaIDE [118] Eclipse plug-in for OCaml language	Memory-change	Immediately interrupt	Never remove	Developer
WitchDoctor [56] auto-completes manual refactorings	Memory-change	Never interrupt	Immediately remove	Developer
NCrunch [115] runs tests and computes coverage	Memory-change	Unknown†	Immediately remove	Full
DocMLET [38] Eclipse plug-in for $\LaTeX$ language	Memory-change	Unknown†	Immediately remove	Developer
dLabPro [37] Eclipse plug-in for dLabPro language	Memory-change	Unknown†	Immediately remove	Developer
Embedded CAL [52] embeds CAL language into Java	Memory-change	Unknown†	Immediately remove	Developer
Eclipse continuous testing [130]	Memory-change	Unknown†	Immediately remove	Developer
Sureassert UC [143] runs tests and computes coverage	Memory-change	Unknown†	Immediately remove	Developer
Blueprint [17] searches code examples from the Internet	Memory-change	Unknown†	Unknown†	Unknown†
Forms/3 [149] evaluates and visualizes the source code	Memory-change	Unknown†	Unknown†	Unknown†
Lighthouse [96] summarizes overall development effort	File-change	Unknown†	Immediately remove	Developer
WeCode [69] detects collaboration conflicts	File-change	Unknown†	Unknown†	Full
Eclipse incremental compiler [46]	Other (build)	Immediately interrupt	Immediately remove	Developer
FindBugs [55] defect detector	Other (build)	Immediately interrupt	Immediately remove	Developer
Checkstyle [43] detects code style violations	Other (build)	Never interrupt	Immediately remove	Developer
Infinitest [78] runs tests	Other (build)	Never interrupt	Never remove	Developer
TypeScript [145] Eclipse plug-in for TypeScript language	Other (build)$^\diamond$	Never interrupt	Immediately remove	Developer
TSLint [144] lints type script code	Other (build)	Never interrupt	Immediately remove	Developer
SConsolidator [134] Eclipse plug-in for SCons build system	Other (build)	Unknown†	Immediately remove	Developer
JUnitLoop [85] runs test	Other (build)	Unknown†	Immediately remove	Developer
InPlace Activator [79] activates and updates source plug-ins	Other (build)	Unknown†	Immediately remove	Developer
GoClipse [66] Eclipse plug-in for Go language	Other (build)	Unknown†	Immediately remove	Developer
EclipseFP [51] Eclipse plug-in for Haskell language	Other (build)	Unknown†	Immediately remove	Developer
JDE [81] Eclipse plug-in for BlackBerry Java language	Other (build)	Unknown†	Immediately remove	Developer
CAL [25] Eclipse plug-in for CAL language	Other (build)	Unknown†	Immediately remove	Developer
JSON Schema Validation [84] validates JSON files	Other (build)	Unknown†	Immediately remove	Developer
Metrics [47] such as cyclomatic complexity	Other (build)	Unknown†	Unknown†	Unknown†
Visual Studio continuous testing [34]	Other (build)	Unknown†	Unknown†	Unknown†
CDT [110] detects likely database update errors	Other (database)	Never interrupt	Never remove	Developer
Crystal [21] detects collaboration conflicts	Periodic (10 m)	Never interrupt	Never remove	Full
APE [3] Eclipse plug-in for AnsProg programming environment	Manual$^\diamond$	N/A	N/A	N/A
ML-Dev [104] Eclipse plug-in for Standard ML language	Manual$^\diamond$	N/A	N/A	N/A
EMFText [53] Eclipse plug-in for Ecore metamodel	Manual$^\diamond$	N/A	N/A	N/A
Bio-PEPA [9] Eclipse plug-in for Bio-PEPA language	Manual$^\triangledown$	N/A	N/A	N/A
Hibernate Synchronizer [73] Eclipse plug-in for Hibernate framework	Manual$^\triangledown$	N/A	N/A	N/A
n-gram-based code completion [74]	Unknown†	Unknown†	Unknown†	Unknown†

* The analysis is 0.5 seconds delayed. For other analyses, the documentation does not describe a delay.
† The publications and tool documentation do not give adequate information to categorize the tool.
$^\diamond$ In addition to the main analysis listed in the table, the plug-in includes secondary memory-change-triggered continuous analyses.
$^\triangledown$ In addition to the main analysis listed in the table, the plug-in includes secondary file-change-triggered continuous analyses.

Figure 6.1: Previous continuous analysis tools, categorized according to the design dimensions of Section 3.2. The first six tools are ε-continuous. The continuous IDE plug-ins for language extensions provide parsing, compilation, auto-complete, quick fix, and/or refactoring support. "Developer" isolation means that the developer is isolated from the changes made by an impure analysis, but the analysis is not isolated from the developer's changes; this is adequate for building only *pure* ε-continuous analysis tools. For IDEs that support auto-build, build-triggered analyses are equivalent to file-change-triggered analyses.

conclude that developers prefer to build ε-continuous tools for at least some analyses, but that the effort required to build such tools prevents their development.

Building an ε-continuous analysis without Codebase Analysis is prohibitively difficult and results in poor designs. As an example, an earlier Eclipse continuous testing plug-in [130] is ε-continuous, but making it ε-continuous required hacking into the core Eclipse plug-ins, so it does not work with subsequent versions of Eclipse. As another example, to achieve isolation, Quick Fix Scout [109] embeds and maintains its own copy codebase in the developer's workspace, significantly complicating its design and implementation. Further, embedding replication logic inside the analysis makes it difficult to debug the replication logic, as bugs that break the synchronization between the copy codebase and the developer's codebase are difficult to isolate. In contrast, as Section 3.5.2 has argued, Solstice makes it easier to write Eclipse-integrated analyses that maintain isolation and currency.

6.2 Speculative Analysis

Speculative analyses are impure; a speculative analysis modifies the codebase before computing its results. A continuous speculative analysis cannot run on the developer's codebase since the intermediate modifications done by this analysis would confuse the developer. To prevent such confusion, a continuous speculative analysis either incrementally maintains a copy of the developer's codebase or creates a copy of the developer's codebase before each analysis execution.

Modern IDEs have limited support for impure analyses. For example, Eclipse provides working copies [50], an AST node copy of a compilation unit. Using working copies, it is possible to make modifications to a compilation unit, without changing the codebase. However, an offline impure analysis needs to go through substantial modifications to create and modify these working copies, rather than the codebase on the disk. Codebase Analysis supports impure analyses without changes to the underlying offline analysis. The impure analysis runs on and modifies the copy codebase, which has no effect on the developer's codebase.

Previously, Brun et al. [21, 20, 22] created a speculative analysis for proactively detecting collaboration conflicts for decentralized VCSs, called Crystal. To compute pairwise collaboration conflicts, Crystal [20] clones the developer's VCS repository into a temporary location and periodically compares this copy repository with the other developers' repositories. When two repositories do not conflict, Crystal attempts to build and test the merged codebase, to compute build and test conflicts. As an extension of this work, we built Beacon [7] for centralized VCSs and larger codebases. Guimarães and Silva [69] proposed a technique that continuously merges committed and uncommitted code to create a codebase, that approximates a software's global state, which can be built and analyzed to detect conflicts before check-in.

6.2.1 Speculative Analysis of Integrated Development Environment Recommendations

The interest in software recommendation systems — "software … that provides information items estimated to be valuable for a software engineering task in a given context" [126] — has grown over the past few years, with an increasing number of research results and tools, as well as an ongoing workshop [75].

Work in recommendation systems includes: defining recommendations for new domains, such as requirements elicitation [27] and team communication [151]; frameworks for defining recommendation systems [99]; and techniques for choosing recommendations to include in a system, such as data mining [133].

Some efforts that are more directly relevant to Quick Fix Scout also aim to improve IDE recommendations. Robbes and Lanza propose eight different ways to reorder code-completion recommendations; they evaluated these techniques on a realistic benchmark, and show that reordering the matches based on historical usage provides the greatest improvement [125]. Bruch et al. reorder and filter the Eclipse auto-complete dialog using mined historical data and developer usage habits [18]. Recently, Perelman et al. showed that Visual Studio auto-complete (IntelliSense) can be improved by using developer-supplied partial types, to search all APIs for auto-completions that would transform the input type to the expected output type [121].

In contrast to the first two of these approaches, which rely on past usage patterns, Quick Fix Scout reorders recommendations based on information about properties of the program that will be created if a recommendation is selected. In contrast to the third approach, the developer need not add any information to the program for Quick Fix Scout (or, of course, Quick Fix) to work. In addition, we have recently shown how to improve IDE recommendations by considering the interactions between existing recommendations [108].

In addition to industrial efforts related to Quick Fix,[1] some research efforts address various aspects of Quick Fix. For example, a paper on automatic refactoring in the face of problems such as references to unavailable declarations mentions an experimental participant's idea to augment the system with invocations to Quick Fix [86]. As another example, a recommendation system approach to increasing reuse also suggests integrating their system through Quick Fix [80].

[1] `http://eclipse.org/recommenders`

Quick Fix Scout is built on speculative analysis: a technique that computes precise information about likely future states of a program and presents this information to the developer so that she can make better and more informed decisions [19]. Applying speculative analysis on collaborative software development [21], Brun et al. built Crystal [20]: a tool that notifies developers as soon as a conflict emerges. The biggest difference between Crystal and Quick Fix Scout is the granularity of the speculation. For collaboration conflicts, it is acceptable if the developer is notified after the conflict emerges since she would still be able to find the reason of the conflict and coordinate it. As a result, Crystal does not have to work on the most recent copy of the project and might report results with some delay. However, when a developer invokes Quick Fix, she needs the results for the recent version of the project as the results from any previous version is not acceptable and actionable. In addition, developers want to see the results as soon as the Quick Fix dialog is created since it takes a couple of seconds for them to decide what to choose.

6.3 Fine-grained History Frameworks

The typical way to create development histories is by using version control systems (VCSs), such as Subversion [32], Mercurial [101], and Git [59]. Unlike Codebase Manipulation, these systems are manual and the history they provide has a fixed, typically coarse granularity. Developers may change the filesystem state to earlier snapshots in the history, and may compare the differences between two snapshots, but cannot easily alter the history to suit particular development tasks.

VCSs require the developer to manually create each snapshot. Developers frequently forget to create snapshots, or simply do not know the best time to to so. As a result, the development history is often coarse-grained or incomplete. For example, a single edit may include changes relevant to multiple development tasks, and changes developers make but overwrite before creating a snapshot are lost. This makes VCS histories suboptimal for many analyses or manual inspection. Codebase Manipulation addresses these limitations by automatically recording the history of *all* edits and providing the framework for rewriting this history into custom granularities better suited for development tasks.

Some VCSs allow limited history rewriting [63, 102]. For example, `git rebase` can collapse, expand, swap, and remove edits [64]. However, these tools are complex, prevent collaboration because rewriting a shared history prevents subsequent sharing, and are irreversible and lead to further history information loss. By contrast, Codebase Manipulation history transformations are high-level, which hides all internal complexity, reversible, and keep intact the recorded history's integrity to enable collaboration.

Development histories can also be created automatically by recording developer actions. Fluorite [153] stores fine-grained edits to visualize, replay, and query the development history, and implements fine-grained selective undo [26]. Built on Fluorite, Azurite studies developers' backtracking patterns [154] and also enables selective undo [155]. CodingSpectator [117] and CodingTracker record and use the fine-grained development history to study refactoring practices [146], development practices [117], and fine-grained change patterns [116]. Storyteller VCS uses the fine-grained history to transfer knowledge from an experienced developer to an inexperienced one [98]. IDE++ [67, 68] maintains a fine-grained development history to improve development by analyzing fine-grained code changes. Each of these tools focuses on particular development tasks or research goals. As a result, these automatically recorded fine-grained histories are inflexible and only suitable for the tasks that require their particular granularity. By contrast, Codebase Manipulation is applicable to many tasks because it records a flexible history whose granularity can be transformed to match each particular task.

To aid understanding how a history should be rewritten, heuristics can detect related changes to help identify which changes in a large edit may need to be untangled. These heuristics include historical code change patterns [92] and change couplings, data dependencies, and code metrics [72]. These approaches focus on detangling large edits, which is a problem of manually-recorded histories, but are complementary to our Untangler tool (recall Section 5.4.1) and can help automate selecting which edits Untangler should collapse. Meanwhile Codebase Manipulation provides access to overwritten changes, potentially improving the effectiveness of these tools.

Development histories simplify some software engineering tasks. For example, Git's `annotate` [60] and `blame` [62] commands can help understand the context of an earlier change, and test bisection [61] and delta debugging [156, 157] can help find the cause of a regression failure. However, the history's granularity affects the effectiveness of these tools. Codebase Manipulation is complementary to these tools and can improve their effectiveness by transforming the granularity into one most suitable for the task. Further, because Codebase Manipulation automatically records every developer edit, it can create richer history views of more granularities than is possible with manually-created histories, further improving tool effectiveness.

Mining software repositories research uses development histories to understand development practices [21, 22, 158], to localize bugs [114, 97, 113, 119, 103, 100], and to help collaborative teams work together [10]. However, performing analyses on manually-recorded histories may lead to incorrect conclusions [11]. A history created by recording the edits at each save operation can be used to visualize development and create development summaries [28] and to study the evolution of students' projects [139]. These repositories are finer-grained and more complete than manually-created ones and research on such repositories has, for example, identified a correlation between static analysis warnings and test failures [140]. The histories created by Codebase Manipulation are finer-grained, richer in terms of containing information about developer actions, and more complete, as they include edits a developer may overwrite before saving a file. This potentially creates better data sets for mining software repositories research.

6.4 Other Uses of Replication

Replication — the idea of using a copy of a resource such as code or data — is well-studied in the context of storage and distributed systems research. Previous work uses replication to increase fault tolerance and performance.

Redundant array of inexpensive disks (RAID) [30, 120] is a system that copies the data on user's hard drive into multiple disks to increase fault tolerance and performance. A RAID system can detect and recover from disk failures. When accessing data, a RAID system reads it from multiple disks. If there is a failure and all disks do not fail in the same way, retrieved values will be inconsistent. A RAID system recovers by assuming that a value retrieved by the majority of the disks is correct.

In distributed systems the Replicated State Machine [94, 132] approach is a widely used method for implementing fault-tolerant services by replicating servers and ordering client requests that execute on these servers. These systems (e.g., SMART [14]) use consensus protocols such as Paxos [95] to order client requests. Apart from the replicated state machine approach, a lot of distributed systems use replication and consensus protocols to achieve fault-tolerance and better performance. These systems include Chubby [24] that uses Paxos to implement a coordination service, Megastore [6] that uses Paxos to replicate primary user data across data centers in a high performance storage system.

Codebase Replication creates and maintains a copy of the developer's codebase to simplify software development tasks. Similar to the previous research, this copy codebase is eventually consistent with the developer's codebase. However, Codebase Replication does not treat the developer's codebase and the copy codebase equally. The flow of information is almost always from the developer's codebase to the copy codebase. The computations ran on the copy codebase do not run on the developer's codebase. The modifications on the copy codebase are not replicated on the developer's codebase. Codebase Replication can be extended to maintain multiple copies of the codebase: one master and many slaves. In this extended design, Codebase Replication can use redundant copies to increase fault tolerance (e.g., if an analysis makes a copy inaccessible) or increase performance (e.g., by running multiple analyses in parallel on multiple copies). Maintaining perfect replicas to increase fault tolerance and performance is left as future work.

Chapter 7
CONTRIBUTIONS

This dissertation shows how an incrementally maintained copy of the developer's codebase can enhance software development. The dissertation first introduces Codebase Replication, a framework that creates and maintains a copy of the developer's codebase. Codebase Replication lets other frameworks run programs on the copy codebase without interrupting development. The dissertation focuses on three frameworks built on Codebase Replication. First, a continuous analysis framework like Codebase Analysis improves developers' interaction with the current codebase through continuous analysis feedback. By implementing four continuous analyses in less than 710 lines of code and 20 hours on average, we showed that Codebase Analysis makes it easy to build IDE-integrated continuous analyses [105]. In a case study on 10 developers, we showed that developers liked continuous feedback and Codebase Analysis continuous testing, and they did not perceive additional overhead [106]. Second, speculative analyses can enhance development by exploring potential future states of the software. Codebase Analysis is designed to support speculative analyses. Applying speculative analysis in the context of Eclipse Quick Fixes, we implemented Quick Fix Scout. In a controlled experiment, we showed that Quick Fix Scout speeds up error-removal tasks by 10% [109]. Third, a history manipulation framework like Codebase Manipulation simplifies information-retrieval tasks from the development history. We showed that Codebase Manipulation can express three distinct tasks: untangling code changes, pinpointing cause of regression failures, and understanding software evolution. We prototyped Codebase Manipulation in Bread. Bread maintains a fine-grained history in real time with low overhead. Using Bread, we implemented Untangler for untangling code changes, and Bisector for pinpointing the cause of regression bugs. Our initial experience with these tools suggests that Bread can simplify information-retrieval tasks.

BIBLIOGRAPHY

[1] Apache Ant. http://ant.apache.org/. Accessed on January 29, 2015.

[2] Apache Maven. http://maven.apache.org/. Accessed on January 29, 2015.

[3] AnsProlog programming environment. https://github.com/robibbotson/APE/. Accessed on February 13, 2015.

[4] Robert Arnold and Shawn Bohner. *Software Change Impact Analysis*. Wiley-IEEE Computer Society Press, July 1996.

[5] Extended ASM, a byte code manipulator. Distributed as part of Annotation File Utilities. http://types.cs.washington.edu/annotation-file-utilities/. Accessed on September 21, 2014.

[6] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *the 5th Conference on Innovative Data Systems Research*, CIDR'11, pages 223–234, Asilomar, CA, USA, January 2011.

[7] Beacon. http://blogs.msdn.com/b/msr_er/archive/2011/09/07/seif-project-crystal-receives-acm-sigsoft-distinguished-paper-award.aspx, 2011.

[8] Keith H. Bennett and Vaclav Rajlich. Software maintenance and evolution: A roadmap. In *the Future of Software Engineering*, pages 73–87, Limerick, Ireland, 2000.

[9] An Eclipse plug-in supporting the Bio-PEPA domain specific language. https://github.com/Bio-PEPA/Bio-PEPA/. Accessed on February 13, 2015.

[10] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? An empirical case study of Windows Vista. In *the 31st International Conference on Software Engineering*, ICSE'09, pages 518–528, Vancouver, BC, Canada, May 2009.

[11] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining Git. In *the 6th Working Conference on Mining Software Repositories*, MSR'09, pages 1–10, Vancouver, BC, Canada, May 2009.

[12] Barry W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[13] Caspar Boekhoudt. The Big Bang Theory of IDEs. *Queue*, 1(7):74–82, October 2003.

[14] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *the 8th Symposium on Networked Systems Design and Implementation*, NSDI'11, Boston, MA, USA, March 2011.

[15] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. Code Bubbles: Rethinking the user interface paradigm of integrated development environments. In *the 32nd International Conference on Software Engineering*, ICSE'10, pages 455–464, Cape Town, South Africa, May 2010.

[16] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. A research demonstration of Code Bubbles. In *the 32nd International Conference on Software Engineering*, ICSE'10, pages 293–296, Cape Town, South Africa, May 2010.

[17] Joel Brandt, Mira Dontcheva, Marcos Weskamp, and Scott R. Klemmer. Example-centric programming: Integrating web search into the development environment. In *the 28th Conference on Human Factors in Computing Systems*, CHI'10, pages 513–522, Atlanta, GA, USA, April 2010.

[18] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *the 7th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE'09, pages 213–222, Amsterdam, The Netherlands, August 2009.

[19] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative analysis: Exploring future states of software. In *the Workshop on the Future of Software Engineering Research*, FoSER'10, pages 59–63, Santa Fe, NM, USA, November 2010.

[20] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Crystal: Precise and unobtrusive conflict warnings. In *the 8th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering Tool Demonstration Track*, ESEC/FSE'11, pages 444–447, Szeged, Hungary, September 2011.

[21] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *the 8th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE'11, Szeged, Hungary, September 2011.

[22] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE Transactions on Software Engineering (TSE)*, 39(10):1358–1375, October 2013.

[23] M. M. Burnett, J. W. Atwood Jr., and Z. T. Welch. Implementing level 4 liveness in declarative visual programming languages. In *the Symposium on Visual Languages*, VL'98, pages 126–133, September 1998.

[24] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *the 7th Symposium on Operating Systems Design and Implementation*, OSDI'06, pages 335–350, Seattle, WA, USA, November 2006.

[25] An Eclipse plug-in supporting the CAL domain specific language. https://github.com/levans/Embedded-CAL/. Accessed on February 13, 2015.

[26] Aaron G. Cass and Chris S. T. Fernandes. Modeling dependencies for cascading selective undo. In *the 10th Conference on Human-Computer interaction*, Rome, Italy, September 2005.

[27] Carlos Castro-Herrera, Chuan Duan, Jane Cleland-Huang, and Bamshad Mobasher. A recommender system for requirements elicitation in large-scale software projects. In *the Symposium on Applied Computing*, SAC'09, pages 1419–1426, 2009.

[28] Jacky Chan, Alan Chu, and Elisa Baniassad. Supporting empirical studies by non-intrusive collection and visualization of fine-grained revision history. In *the 5th Eclipse Technology Exchange Workshop*, eTX'07, pages 60–64, Montreal, QC, Canada, October 2007.

[29] Check synchronization. http://www.cs.umd.edu/class/fall2004/cmsc433/checkSync.html. Accessed on September 21, 2014.

[30] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, reliable secondary storage. *Computing Surveys*, 26(2):145–185, June 1994.

[31] Code recommenders. http://www.eclipse.org/recommenders/. Accessed on September 21, 2014.

[32] Ben Collins-Sussman. The Subversion project: Building a better CVS. *Linux Journal*, 2002(94):3, February 2002.

[33] Continuous analysis. http://www.klocwork.com/products/documentation/current/Continuous_analysis. Accessed on January 26, 2015.

[34] Continuous testing for Visual Studio. http://ox.no/software/continuoustesting/. Accessed on September 21, 2014.

[35] Crossword Sage. http://sourceforge.net/projects/crosswordsage/. Accessed on September 21, 2014.

[36] Robert DeLine and Kael Rowan. Code Canvas: Zooming towards better development environments. In *the 32nd International Conference on Software Engineering*, ICSE'10, pages 207–210, Cape Town, South Africa, May 2010.

[37] An Eclipse plug-in supporting the dLabPro domain specific language. https://github.com/matthias-wolff/dLabPro-Plugin/. Accessed on February 13, 2015.

[38] An Eclipse plug-in supporting the $\LaTeX$ domain specific language. https://github.com/walware/docmlet/. Accessed on February 13, 2015.

[39] Eclipse. http://www.eclipse.org/. Accessed on September 21, 2014.

[40] Eclipse API: IDocumentListener. http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/jface/text/IDocumentListener.html. Accessed on October 2, 2014.

[41] Eclipse API: IResourceChangeListener. http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/IResourceChangeListener.html. Accessed on October 2, 2014.

[42] Eclipse project builders and natures. http://www.eclipse.org/articles/Article-Builders/builders.html. Accessed on September 21, 2014.

[43] Eclipse-Checkstyle integration. http://eclipse-cs.sourceforge.net/. Accessed on September 21, 2014.

[44] Eclipse: How do I use a model reconciler? https://wiki.eclipse.org/FAQ_How_do_I_use_a_model_reconciler%3F. Accessed on January 31, 2015.

[45] Eclipse: Java compile errors/warnings preferences. http://help.eclipse.org/topic/org.eclipse.jdt.doc.user/reference/preferences/java/compiler/ref-preferences-errors-warnings.htm. Accessed on September 21, 2014.

[46] Eclipse: JDT core component. http://www.eclipse.org/jdt/core/index.php. Accessed on September 21, 2014.

[47] Eclipse Metrics plug-in. http://sourceforge.net/projects/metrics/. Accessed on September 21, 2014.

[48] Eclipse: The jobs API. http://www.eclipse.org/articles/Article-Concurrency/jobs-api.html. Accessed on September 21, 2014.

[49] Eclipse: Views. http://help.eclipse.org/topic/org.eclipse.platform.doc.isv/reference/extension-points/org_eclipse_ui_views.html. Accessed on September 21, 2014.

[50] Eclipse: What is a working copy? http://wiki.eclipse.org/FAQ_What_is_a_working_copy%3F. Accessed on September 21, 2014.

[51] An Eclipse plug-in supporting the Haskell domain specific language. https://github.com/JPMoresmau/eclipsefp/. Accessed on February 13, 2015.

[52] Embedded CAL. https://github.com/levans/Embedded-CAL/. Accessed on February 13, 2015.

[53] An Eclipse plug-in supporting the Ecore metamodel. https://github.com/DevBoost/EMFText/. Accessed on February 13, 2015.

[54] eVHDL: Eclipse plug-in for developing VHDL code. https://github.com/HepaxCodex/eVHDL/. Accessed on October 2, 2014.

[55] FindBugs. http://findbugs.sourceforge.net/. Accessed on September 21, 2014.

[56] Stephen R. Foster, William G. Griswold, and Sorin Lerner. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 222–232, Zurich, Switzerland, June 2012.

[57] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. Reconciling manual and automatic refactoring. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 211–221, Zurich, Switzerland, June 2012.

[58] Xi Ge and Emerson Murphy-Hill. BeneFactor: A flexible refactoring tool for Eclipse. In *the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'11, pages 19–20, Portland, OR, USA, October 2011.

[59] Git. http://www.git-scm.com/. Accessed on September 21, 2014.

[60] Git annotate. https://www.kernel.org/pub/software/scm/git/docs/git-annotate.html. Accessed on September 21, 2014.

[61] Git bisect. https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html. Accessed on September 21, 2014.

[62] Git blame. https://www.kernel.org/pub/software/scm/git/docs/git-blame.html. Accessed on September 21, 2014.

[63] Git: History rewriting. http://git-scm.com/book/en/Git-Tools-Rewriting-History. Accessed on September 21, 2014.

[64] Git: Rebase. http://www.git-scm.com/book/en/Git-Branching-Rebasing. Accessed on September 21, 2014.

[65] David Samuel Glasser. Test factoring with `amock`: Generating readable unit tests from system tests. Master's thesis, Massachusetts Institute of Technology, Boston, MA, USA, August 2007.

[66] An Eclipse plug-in supporting the Go domain specific language. https://github.com/GoClipse/goclipse/. Accessed on February 13, 2015.

[67] Zhongxian Gu. Capturing and exploiting fine-grained IDE interactions. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 1630–1631, Zurich, Switzerland, June 2012.

[68] Zhongxian Gu. *Toward Effective Debugging by Capturing and Reusing Knowledge*. PhD thesis, University of California, Davis, Davis, CA, USA, 2013.

[69] Mário Luís Guimarães and António Rito Silva. Improving early detection of software merge conflicts. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 342–352, Zurich, Switzerland, June 2012.

[70] Masatomo Hashimoto and Akira Mori. Diff/TS: A tool for fine-grained structural change analysis. In *the 15th Working Conference on Reverse Engineering*, WCRE'08, pages 279–288, Antwerp, Belgium, October 2008.

[71] Peter Henderson and Mark Weiser. Continuous execution: The VisiProg environment. In *the 8th International Conference on Software Engineering*, ICSE'85, pages 68–74, London, England, August 1985.

[72] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *the 6th Working Conference on Mining Software Repositories*, MSR'09, pages 121–130, San Francisco, CA, USA, May 2013.

[73] Hibernate Synchronizer. https://github.com/jhudson8/hibernate-synchronizer/. Accessed on February 13, 2015.

[74] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 837–847, Zurich, Switzerland, June 2012.

[75] Reid Holmes, Martin Robillard, Rob Walker, Tom Zimmermann, and Walid Maalej. International workshops on recommendation systems for software engineering (RSSE). https://sites.google.com/site/rsseresearch, 2012.

[76] David Hovemeyer and William Pugh. Finding bugs is easy. In *the 19th Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'04, pages 132–136, Vancouver, BC, Canada, October 2004.

[77] Hudson. http://hudson-ci.org/. Accessed on September 21, 2014.

[78] Infinitest. http://infinitest.github.io/. Accessed on September 21, 2014.

[79] InPlace Activator. https://github.com/eirikg/no.javatime.inplace/. Accessed on February 13, 2015.

[80] Werner Janjic, Dietmar Stoll, Philipp Bostan, and Colin Atkinson. Lowering the barrier to reuse through test-driven search. In *the Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, SUITE'09, pages 21–24, 2009.

[81] An Eclipse plug-in supporting the Blackberry Java domain specific language. https://github.com/blackberry/Eclipse-JDE/. Accessed on February 13, 2015.

[82] Jenkins. http://jenkins-ci.org/. Accessed on September 21, 2014.

[83] Lustre plug-in for Eclipse with JKind analysis support. https://github.com/agacek/jkind-xtext/. Accessed on October 2, 2014.

[84] JSON Schema Validation. https://github.com/sabina-jung/JSON-Schema-Validation-Eclipse/. Accessed on February 13, 2015.

[85] JUnitLoop. https://github.com/DevBoost/JUnitLoop/. Accessed on February 13, 2015.

[86] Puneet Kapur, Brad Cossette, and Robert J. Walker. Refactoring references for library migration. In *the Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'10, pages 726–738, Reno/Tahoe NV, USA, October 2010.

[87] Raghu R. Karinthi and Mark Weiser. Incremental re-execution of programs. In *Symposium on Interpreters and Interpretive Techniques*, SIIT'87, pages 38–44, St. Paul, MN, USA, 1987.

[88] Harry Katzan Jr. Batch, conversational, and incremental compilers. In *the American Federation of Information Processing Societies*, AFIPS'69, pages 47–56, Boston, MA, USA, May 1969.

[89] Mik Kersten and Gail C. Murphy. Mylar: A degree-of-interest model for IDEs. In *the 4th International Conference on Aspect-oriented Software Development*, AOSD'05, pages 159–168, Chicago, IL, USA, March 2005.

[90] Mik Kersten and Gail C. Murphy. Using task context to improve programmer productivity. In *the 14th Symposium on the Foundations of Software Engineering*, FSE'06, pages 1–11, Portland, OR, USA, November 2006.

[91] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. Predicting faults from cached history. In *the 29th International Conference on Software Engineering*, ICSE'07, pages 489–498, Minneapolis, MN, USA, May 2007.

[92] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! Are you committing tangled changes? In *the 22nd International Conference on Program Comprehension*, ICPC'14, Hyderabad, India, June 2014.

[93] Shuvendu K. Lahiri, Kapil Vaswani, and Charles A. R. Hoare. Differential static analysis: Opportunities, applications, and challenges. In *the Workshop on the Future of Software Engineering Research*, FoSER'10, pages 201–204, Santa Fe, NM, USA, November 2010.

[94] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, July 1978.

[95] Leslie Lamport. The part-time parliament. *Transactions on Computer Systems*, 16(2):133–169, May 1998.

[96] Lighthouse. https://github.com/uci-sdcl/lighthouse/. Accessed on February 13, 2015.

[97] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *the 10th the European Software Engineering Conference, held jointly with the 13th Symposium on the Foundations of Software Engineering*, ESEC/FSE'05, pages 296–305, Lisbon, Portugal, September 2005.

[98] Mark Mahoney. The Storyteller version control system: Tackling version control, code comments, and team learning. In *the 3rd Conference on Systems, Programming, Languages and Applications: Software for Humanity*, SPLASH'12, pages 17–18, Tucson, AZ, USA, October 2012.

[99] Felipe Martins Melo and Álvaro Pereira Jr. A component-based open-source framework for general-purpose recommender systems. In *the 14th Symposium on Component Based Software Engineering*, CBSE'11, pages 67–72, 2011.

[100] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, December 2010.

[101] Mercurial. http://mercurial.selenic.com/. Accessed on September 21, 2014.

[102] Mercurial: Editing history. http://mercurial.selenic.com/wiki/EditingHistory. Accessed on September 21, 2014.

[103] Ayse Tosun Misirli, Ayse Basar Bener, and Resat Kale. AI-based software defect predictors: Applications and benefits in a case study. *AI Magazine*, 32(2):57–68, 2011.

[104] An Eclipse plug-in supporting the standard ML language. https://github.com/andriusvelykis/ml-dev/. Accessed on February 13, 2015.

[105] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Making offline analyses continuous. In *the 9th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE'13, pages 323–333, Saint Petersburg, Russia, August 2013.

[106] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Reducing Feedback Delay of Software Development Tools via Continuous Analyses (in press). *IEEE Transactions on Software Engineering (TSE)*, 2015.

[107] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Quick Fix Scout. http://quick-fix-scout.googlecode.com/. Accessed on September 21, 2014.

[108] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Improving IDE Recommendations by Considering Global Implications of Existing Recommendations. In *the 34th International Conference on Software Engineering, New Ideas and Emerging Results Track*, ICSE'12, pages 1349–1352, Zurich, Switzerland, June 2012.

[109] Kıvanç Muşlu, Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Speculative analysis of integrated development environment recommendations. In *the 3rd Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA'12, pages 669–682, Tucson, AZ, USA, October 2012.

[110] Kıvanç Muşlu, Yuriy Brun, and Alexandra Meliou. Data debugging with continuous testing. In *the 9th joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering, New Ideas Track*, ESEC/FSE'13, pages 631–634, Saint Petersburg, Russia, August 2013.

[111] Kıvanç Muşlu, Luke Swart, Alain Orbino, Yuriy Brun, Michael D. Ernst, and David Notkin. Solstice. https://bitbucket.org/kivancmuslu/solstice/. Accessed on September 21, 2014.

[112] Gail C. Murphy, Mik Kersten, and Leah Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[113] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *the 28th International Conference on Software Engineering*, ICSE'06, pages 452–461, Shanghai, China, May 2006.

[114] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *the 21st International Symposium on Software Reliability Engineering*, ISSRE'10, pages 309–318, San Jose, CA, USA, November 2010.

[115] NCrunch. http://www.ncrunch.net/. Accessed on January 26, 2015.

[116] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *the 36th International Conference on Software Engineering*, ICSE'14, Hyderabad, India, June 2014.

[117] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 79–103, Beijing, China, June 2012.

[118] OcaIDE: OCaml plug-in for Eclipse. https://github.com/nbros/OcaIDE/. Accessed on October 2, 2014.

[119] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *International Symposium on Software Testing and Analysis*, ISSTA'04, pages 86–96, Boston, MA, USA, July 2004.

[120] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *SIGMOD International Conference on Management of Data*, SIGMOD'88, pages 109–116, Chicago, IL, USA, June 1988.

[121] Daniel Perelman, Sumit Gulwani, Tom Ball, and Dan Grossman. Type-Directed completion of partial expressions. In *the Conference on Programming Language Design and Implementation*, PLDI'12, Beijing, China, June 2012.

[122] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. Differential symbolic execution. In *the 16th Symposium on the Foundations of Software Engineering*, FSE'08, pages 226–237, Atlanta, GA, USA, November 2008.

[123] PMD. http://pmd.sourceforge.net/. Accessed on September 21, 2014.

[124] Ganesan Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *the 20th Symposium on Principles of Programming Languages*, POPL'93, pages 502–510, Charleston, SC, USA, January 1993.

[125] Romain Robbes and Michele Lanza. How program history can improve code completion. In *the 23rd International Conference on Automated Software Engineering*, ASE'08, pages 317–326, L'Aquila, Italy, September 2008.

[126] Martin Robillard, Robert Walker, and Thomas Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27:80–86, 2010.

[127] Izzet Safer, Gail C. Murphy, Julie Waterhouse, and Jin Li. A focused learning environment for Eclipse. In *the 4th Eclipse Technology Exchange Workshop*, eTX'06, pages 75–79, Portland, OR, USA, October 2006.

[128] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *the 14th International Symposium on Software Reliability Engineering*, ISSRE'03, pages 281–292, Denver, CO, USA, November 2003.

[129] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *International Symposium on Software Testing and Analysis*, ISSTA'04, pages 76–85, Boston, MA, USA, July 2004.

[130] David Saff and Michael D. Ernst. Continuous testing in Eclipse. In *the 27th International Conference on Software Engineering*, ICSE'05, pages 668–669, St. Louis, MO, USA, May 2005.

[131] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *Transactions on Computer Systems*, 15(4):391–411, November 1997.

[132] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *Computing Surveys*, 22(4):299–319, December 1990.

[133] Kurt Schneider, Stefan Gärtner, Tristan Wehrmaker, and Bernd Brügge. Recommendations as learning: From discrepancies to software improvement. In *the International Workshop on Software Recommendation Systems*, RSSE'12, pages 31–32, 2012.

[134] SConsolidator. https://github.com/IFS-HSR/Sconsolidator/. Accessed on February 13, 2015.

[135] An Eclipse plug-in supporting the Scribble domain specific language. https://github.com/glozachm/scribble-plug-in/. Accessed on October 2, 2014.

[136] Ajeet Shankar and Rastislav Bodík. Ditto: Automatic incrementalization of data structure invariant checks (in Java). In *the Conference on Programming Language Design and Implementation*, PLDI'07, pages 310–319, San Diego, CA, USA, June 2007.

[137] Smart save plug-in. http://marketplace.eclipse.org/content/smart-save. Accessed on September 21, 2014.

[138] SonarQube. http://www.sonarqube.org/. Accessed on January 29, 2015.

[139] Jaime Spacco, David Hovemeyer, and William Pugh. An Eclipse-based course project snapshot and submission system. In *the 2nd Eclipse Technology Exchange Workshop*, eTX'04, pages 52–56, Barcelona, Spain, March 2004.

[140] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *the 2nd Working Conference on Mining Software Repositories*, MSR'05, pages 1–5, St. Louis, MO, USA, May 2005.

[141] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *the Conference on Programming Language Design and Implementation*, PLDI'07, pages 112–122, San Diego, CA, USA, June 2007.

[142] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *the 19th Australian Computer Science Conference*, pages 582–591, Melbourne, Australia, January 1996.

[143] Sureassert UC. http://www.sureassert.com/uc/. Accessed on January 26, 2015.

[144] TSLint. https://github.com/palantir/eclipse-tslint/. Accessed on February 13, 2015.

[145] An Eclipse plug-in supporting the TypeScript domain specific language. https://github.com/palantir/eclipse-typescript/. Accessed on February 13, 2015.

[146] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *the 34th International Conference on Software Engineering*, ICSE'12, pages 233–243, Zurich, Switzerland, June 2012.

[147] Visual Studio. http://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx. Accessed on September 21, 2014.

[148] Voldemort. http://www.project-voldemort.com/voldemort/. Accessed on September 21, 2014.

[149] E. M. Wilcox, J. W. Atwood Jr., M. M. Burnett, J. J. Cadiz, and C. R. Cook. Does continuous visual feedback aid debugging in direct-manipulation programming systems? In *the Conference on Human Factors in Computing Systems*, CHI'97, pages 258–265, Atlanta, GA, USA, March 1997.

[150] An Eclipse plug-in supporting the NesC domain specific language. https://github.com/mimuw-distributed-systems-group/wNesC-Eclipse-Plug-in/. Accessed on February 13, 2015.

[151] P. F. Xiang, A. T. T. Ying, P. Cheng, Y. B. Dang, K. Ehrlich, M. E. Helander, P. M. Matchen, A. Empere, P. L. Tarr, C. Williams, and S. X. Yang. Ensemble: a recommendation tool for promoting communication in software teams. In *the International Workshop on Recommendation Systems for Software Engineering*, RSSE'08, pages 2–2, 2008.

[152] Xtext. https://www.eclipse.org/Xtext/. Accessed on October 2, 2014.

[153] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In *the 3rd Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU'11, pages 25–30, Portland, OR, USA, October 2011.

[154] YoungSeok Yoon and Brad A. Myers. A longitudinal study of programmers' backtracking. In *the Symposium on Visual Languages and Human-Centric Computing*, VL/HCC'14, pages 101–108, Melbourne, Australia, July 2014.

[155] YoungSeok Yoon and Brad A. Myers. Supporting selective undo in a code editor. In *the 37th International Conference on Software Engineering*, ICSE'15, Firenze, Italy, May 2015.

[156] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *the 7th the European Software Engineering Conference, held jointly with the 7th Symposium on the Foundations of Software Engineering*, ESEC/FSE'99, pages 253–267, Toulouse, France, 1999.

[157] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering (TSE)*, 28(2):183–200, february 2002.

[158] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *the 26th International Conference on Software Engineering*, ICSE'04, pages 563–572, Scotland, UK, May 2004.

VITA

Kıvanç Muşlu is working on software engineering and programming languages. He is advised by Michael Ernst, Yuriy Brun, and David Notkin. Before, he was a student in the Computer Engineering department at Koç University (Istanbul, Turkey) (2005-2010). He was advised by Serdar Taşıran. His research interests are software engineering and programming languages, especially developing techniques and tools to increase programmer productivity and reduce programmer mistakes. Kıvanç graduated from Kabataş Erkek Lisesi (high school) in Spring 2005. He was the 5th best student, over more than 1.8 million students, in the University Entrance Exams (ÖSS) the same year. He received his BSc in Computer Engineering with the valedictorian award from Koç University in July 2010, and his MSc and PhD in Computer Science & Engineering from the University of Washington (Seattle, WA, USA) in December 2012 and June 2015, respectively. Kıvanç was an intern at Microsoft Research, Redmond during Summer 2011 and Summer 2013 working in ESE group and mentored by Christian Bird, Judith Bishop, Tom Zimmermann, Nachiappan Nagappan, and Jacek Czerwonka. Kıvanç was an intern at Facebook Palo Alto during Summer 2012 mentored by Damien Sereni. This fall, Kıvanç will be joining Microsoft's TSE team to empower Microsoft developers to build better and faster software.