TOWARDS MULTI-DIMENSIONAL INTEGRATED DEVELOPMENT ENVIRONMENTS FOR IMPROVED PRODUCTIVITY

BY

YUN YOUNG LEE

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Research Associate Professor Ralph E. Johnson, Chair and Director of Research
Associate Professor Darko Marinov
Professor Grigore Rosu
Associate Professor Sarfraz Khurshid, The University of Texas at Austin
Abstract

Integrated development environments (IDEs) are software applications designed to facilitate writing, building, debugging, and deploying software. IDEs come in many different forms, but they are an integral part of software engineering for many developers; studies reported that 97% of developers use IDEs for their daily development tasks, and 91% of them prefer the modern GUI-based IDEs with a plug-in architecture such as Eclipse and NetBeans. The plug-in architecture offers extensibility for IDEs to support multiple programming languages and allows tool providers and developers to write custom tools as separate plug-ins.

Although plug-ins provide an effective means of extending and customizing IDEs, we believe the very plug-in nature can lead to an IDEs overloaded with plug-ins. A quick search on the Eclipse Marketplace websites lists about 2,000 plug-ins available. In addition, most of these plug-ins provide a distinct functionality that operates separately from other plug-ins or even from the main programming activities, and they also make heavy use of graphical user interfaces because the IDEs themselves are GUI-based. This means that developers not only still have to switch context within an IDE when using these plug-ins, but also have to learn how to use these plug-ins in order to accomplish their tasks. We conjecture that these issues result in counterproductive tools.

We believe, however, that it is possible to build tools that are more intuitive and seamlessly integrated by leveraging developers’ inherent understandings of their code and learned skills in software development processes. This dissertation presents our research effort in creating a new class of plug-ins that addresses these shortcomings without hampering developers’ productivity. Our approach achieves this goal by extending an IDE with new dimensions that allow developers to accomplish their tasks with familiar actions taken in different settings. More specifically, we present two Eclipse plug-ins,
called Drag-and-Drop Refactoring and Tempura, that are designed to address the main problems of plug-in overload for different features; Drag-and-Drop Refactoring adds a new tactile dimension to an IDE’s refactoring tools, and allows developers to bypass complex GUI-based invocation and configuration steps and perform refactorings by moving program elements directly, and Tempura adds a new temporal dimension to code completion and navigation functions in an IDE to allow developers to search for deleted classes, methods, and fields, therefore more intimately integrating version control support into the IDE platform. Our evaluations demonstrate that, compared to widely used substitutes, our tools allow developers to achieve their goals more efficiently and accurately.
To mum.
Acknowledgments

I have received immeasurable and invaluable guidance from many mentors. My advisor, Ralph Johnson, taught me so much about research, software engineering, and teaching. I enjoyed every meeting I had with him for his many stories about software engineering as well as the non-software engineering stories. My former advisor, Darko Marinov, never stopped advising me, for which I am immensely grateful. I also thank Grigore Rosu and Sarfraz Khurshid for graciously agreeing to be on my committee, and for their insight and encouragement.

Six years of my life I spent at the University of Illinois at Urbana-Champaign would have been rather monotonous if not for my friends, Nathan Hirtz, Lucas Cook, Alejandro Gutierrez, Vilas Jagannath, Jeff Overbey, Nicholas Chen, Samira Tasharofi, and Maurice Rabb. Some of them I worked with on various projects, some of them I spent most of days together. But all of them were my support system directly and indirectly. And without them, there certainly wouldn’t have been the Siebel Center Igloo.

My colleague Sam Harwell, and my mentees, Johnston Jiaa, Andrey Zaytsev, and Nikhil Unni, helped with various aspects of my work. I feel that I learned more from them than they from me, but hopefully they had fun at least.

The academic staff at the Computer Science Department have been unbelievably helpful during my time here. Mary Beth Kelly, Rhonda McElroy, Kathy Runck, and Holly Bagwell made this department a second home for me.
To the Daniels family, who has been my family in every sense of the word, I give my deepest love. They were my safe place throughout the years, and I only hope to one day repay the love and support they have shown me.

Without the unconditional and unfaltering love and support of my mum, my sister, and my brother, all this would have been simply impossible.

To Brett Daniel, I thank you. For everything, and always.

Portions of the work presented in this dissertation were supported by the National Science Foundation under Grant Nos. CCF-1213091, CCF-0845628, CCF-0746856, and CCF-1439957.
# Table of Contents

List of Tables .................................................. ix  
List of Figures .................................................. x  

Chapter 1 Introduction ........................................ 1  
1.1 Thesis Statement .......................................... 2  
1.2 Drag-and-Drop Refactoring: New Tactile Dimension for Refactoring Tool's User Interface .................................................. 3  
1.3 Tempura: New Temporal Dimension for Code Completion and Navigation . 6  
1.4 Dissertation Overview .................................... 9  

Chapter 2 Drag-and-Drop Refactoring: Tactile Dimension for Refactoring Invocation and Configuration ........................................ 10  
2.1 DNDRefactoring ............................................. 10  
2.1.1 Motivating Examples .................................... 10  
2.1.2 Design Rationale ....................................... 11  
2.1.3 Tool Features .......................................... 12  
2.1.4 Supporting Floss Refactoring ......................... 16  
2.2 Evaluation Methodology ................................... 17  
2.3 Evaluating Intuitiveness ................................... 20  
2.3.1 Survey Design ......................................... 20  
2.3.2 Results and Observations ............................ 22  
2.4 Evaluating Efficiency and Usability ....................... 23  
2.4.1 Controlled User Study Design ....................... 23  
2.4.2 Controlled User Study Results and Observations .... 24  
2.4.3 Post-Study Qualitative Survey Results ............... 31  
2.5 Limitations .................................................. 32  
2.5.1 Threats to Validity .................................... 32  
2.5.2 C. Limitations of DNDRefactoring .................. 33  

Chapter 3 Tempura: Temporal Dimension for Code Completion and Navigation ........................................ 35  
3.1 Motivating Examples ...................................... 35  
3.2 Tempura Tool .............................................. 36  
3.2.1 Temporal Code Completion ............................ 36  
3.2.2 Temporal Code Navigation ............................ 36  
3.3 Algorithms .................................................. 42  
3.3.1 Checkout Algorithm .................................... 44
3.3.2 Indexing Algorithm ................................................. 45
3.3.3 Challenges .............................................................. 53
3.4 Evaluation ................................................................. 59
  3.4.1 Indexing and Runtime Efficiency ................................ 59
  3.4.2 Controlled User Study ............................................. 61
3.5 Threats to Validity ....................................................... 67

Chapter 4 Related Work ................................................... 69
  4.1 Drag-and-Drop .......................................................... 69
  4.2 Code Completion and Navigation .................................. 70
  4.3 Software Evolution .................................................... 72

Chapter 5 Conclusions and Future Work ............................. 75
  5.1 Future Work ............................................................ 75
    5.1.1 Drag-and-Drop Refactoring ................................. 76
    5.1.2 Tempura .......................................................... 77

References .................................................................. 79
List of Tables

2.1 Refactorings with Drag-and-Drop: within a Java editor................. 13
2.2 Refactorings with Drag-and-Drop: within and between Package Explorer
and Outline View.............................. 14
2.3 Survey results by category. Each column name corresponds to the particu-
lar refactoring asked in the survey............................ 21
2.4 Results of the controlled user study - Configuration Time in seconds.... 25
2.5 Results of the controlled user study - Programming Effort.............. 27
2.6 Results of the controlled user study - Obstacles........................ 30

3.1 Indexing algorithm's indexing of every Java file in each revision from Git
repositories............................................ 60
3.2 Temporal Code Completion Invocations for both algorithms. On-the-fly
algorithm failed to compute code completion for top three classes due to
their long history........................................ 60
3.3 Changes made to LANSimulation project............................... 63
3.4 Questions given to user study subjects.................................. 63
3.5 Grading rubric............................................. 64
3.6 User study results........................................... 65
List of Figures

2.1 Drag-and-drop gestures in (a) Java editor for Extract Method refactoring, and (b) Package Explorer for Extract Type to New File refactoring. 18

2.2 Examples of Survey Questions 19

3.1 Tempura's historical code completion result when invoked on the static LexerATNSimulator type reference. Historical proposals for LexerATNSimulator class are shown in gray, with historical information displayed in tooltip. Example code is from ANTLR4 project. 37

3.2 Selecting a historical code completion proposal opens Eclipse's diff view, comparing the revision that removed the proposal (left) with the previous revision (right). 38

3.3 Tempura's Open Type in History dialog shows historical types, including deleted ones (listed with a strike-through). Selecting a type displays details about the last revision of the type at the bottom of the dialog window. Those types that Tempura identifies to have been renamed (or moved) from another type also describes the change with an arrow. 40

3.4 Tempura's Historical read-only editor with a list of revisions on the left hand side. Blue background color highlights the snippet of code that was changed since the last revision (similarly, green highlights added code). 41

3.5 Type hierarchy of HistoryElement, a data object for storing program elements extracted from history. Each of these data object records the corresponding element's syntactic components. 48

3.6 Indexing algorithm - Parsing 49

3.7 Processing of commits in Indexing Algorithm. It traverses the commits in chronological order, parsing file snapshots in each commit to extract program elements. The extracted program elements are stored as simple data objects indexed by the (enclosing) type's fully-qualified name. 50

3.8 Indexing algorithm - Filtering 52

3.9 Code completion results when invoked on a different new branch in ANTLR4 repository, created following the revision shown in Figure 3.1. Several commits were made in the new branch, during which two static final integer fields, namely TEST and TEST_2, were added and then removed in different revisions. Compared to the code completion results shown in Figure 3.1, Tempura shows the two extra history proposals that pertains only to the new branch. 57
Chapter 1

Introduction

Integrated development environments (IDEs), software applications designed to facilitate software development by providing support features for writing, building, debugging, and deploying software, have been in use since it became possible for developers to write programs via a console or terminal instead of on punch cards [onlf]. Since the introduction of Maestro 1 in the mid 70’s, the world’s first IDE developed by Softlab Munich [onlh], new IDEs have been continuously developed and improved. For example, Hewlett-Packard released Softbench, the first IDE to support a plug-in concept, in 1989 [Lie97], followed by NetBeans by Oracle Cooperation in 1996 [onlb], Eclipse by IBM and IntelliJ by JetBrains in 2001 [onla, onli], among many other IDEs. Nowadays, IDEs have become an indispensable tool for many developers, as surveys show that 97% of developers use IDEs, and of those developers using IDEs, 91% use GUI-based IDEs such as Eclipse and NetBeans [onlc, onld]. While there are some developers who favor command-line oriented tools that use editors such as Emacs and Vim to build IDEs using their standard Unix and GNU build tools (e.g. GNU Compiler Collection and GNU Debugger), or those that use makefiles to manage code building, most of these popular IDEs have a common characteristic, which is that they are a platform for integrating development tools in the form of plug-ins. A plug-in is a smallest unit of IDE function, and a set of plug-ins composes an instance of an IDE. The plug-in nature of these IDEs offers extensibility to support multiple programming languages and allows tool providers and developers to write custom tools as separate plug-ins.

Although plug-ins provide an effective means of extending and customizing IDEs, we believe the very plug-in nature can lead to an overloaded IDE both in terms of user interfaces and operations. Many interactive tools for GUI-based IDEs are heavily dependent
on GUI themselves, and they would likely have their own views or perspectives, each in turn with menus and dialogs for developer input and configuration. For example, many simple refactorings in Eclipse require developers to step through a number of complex selection and configuration steps.

In addition, we believe that quick and easy plug-in development support encourages for too many disparate tools in an IDE. For example, many IDEs have plug-in extensions for popular software development support tools such as version control systems (VCSs). While these plug-ins integrate the external tools into an IDE platform, their operations are still separate from other software development activities like programming and debugging.

These shortcomings force developers to constantly switch context when using plug-ins within an IDE, which we believe counteracts the purpose behind an integrated development environment. We also conjecture that it can discourage developers from using the tools or even IDEs, leading to lower productivity and efficiency. This dissertation presents our research effort in creating a new class of plug-ins that addresses these shortcomings without hampering developers’ productivity by eliminating the context switch between plug-ins. We believe that it is possible to build tools that are more intuitive and instinctive by leveraging developers’ inherent understanding of their code and software development activities.

1.1 Thesis Statement

It is possible to eliminate context switch between plug-ins by leveraging developers' experiences with and inherent understanding of their code, common IDE tools, and other software development activities, and such plug-ins will help improve developers’ productivity. More precisely, we present two plug-ins that make the following claims.

1. Refactorings in IDEs can be made more immediate by using drag-and-drop
gestures for invocation and configuration instead of GUI-heavy dialog boxes, and such approach will benefit developers by putting them in control of refactorings.

2. IDEs can be extended to seamlessly integrate multiple versions of code into a single workspace via common functionality such as code completion and navigation, and such extension will benefit developers by allowing them to learn the history of code first-hand.

To confirm this thesis, this dissertation presents two plug-ins called Drag-and-Drop Refactoring and Tempura that were designed to address the main problems of plug-in overload for different features. Our approach achieves this goal by extending an IDE with new dimensions that allow developers to accomplish their tasks with familiar actions taken in different settings. Drag-and-Drop Refactoring adds a new tactile dimension to an IDE’s refactoring tools, and allows developers to bypass complex GUI-based invocation and configuration steps and perform refactorings by moving program elements directly. Tempura adds a new temporal dimension to code completion and navigation functions in an IDE to allow developers to search for deleted classes, methods, and fields, therefore more intimately integrating VCS support into the IDE platform.

1.2 Drag-and-Drop Refactoring: New Tactile Dimension for Refactoring Tool’s User Interface

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior [Fow99], which aims to improve code readability and maintainability by reducing complexity, helping programmers make design changes during software maintenance. The term refactoring was first introduced by Opdyke and Johnson in early 1990’s, who cataloged and prototyped transformations for object-oriented programs in C++ [OJ90, Opd92]. Shortly thereafter, Brant and Roberts developed the Smalltalk Refactoring Browser, which was inte-
grated into the Smalltalk development environment [RBJ97, Rob99]. Refactoring gained
more popularity following the invention of eXtreme Programming (XP), the first soft-
ware process to promote refactoring as a critical software development activity. Along
with the publication of Fowler’s book which catalogs 72 refactorings for object-oriented
programs [Fow99], refactoring has since become a well-accepted programming practice.

In fact, almost all popular modern IDEs, such as Eclipse, IntelliJ, NetBeans, Visual
Studio, and Xcode, include support for automated refactoring tools. Though no IDE
supports all 72 refactorings that Fowler cataloged in his book [Fow99], the number of
refactorings that IDEs support has only been increasing. For example, Eclipse 2 (as of
2004) supported 14 refactorings but the most recent version of Eclipse (version 4.2) con-
tains 23 refactorings for Java. The current version of NetBeans supports 18 refactorings
and IntelliJ supports more than 30 refactorings.

As automated refactoring tools become more mainstream, there has been much re-
search analyzing their usage patterns. Murphy-Hill et al. analyzed Eclipse refactoring
tool usage and concluded that almost 90% of refactorings are performed manually with-
out the help of the tool [MHPB09]. Our prior work concluded that programmers, on
average, are aware of only eight refactorings in Eclipse [VCN12]. These numbers are
discouraging and suggest that refactoring tools are used infrequently. One of the main
causes behind their disuse is that the current tools suffer from deep usability problems.

Prior research identified at least three dominant usability problems when using auto-
mated refactoring tools [VCN12, MCSW07, OSG05, PGN08, MHAB11, MHB08a]. First,
programmers have trouble identifying opportunities for using the tool. Second, pro-
grammers have difficulty invoking the right refactoring from a lengthy menu of available
refactorings. Programmers often find the names and the position of the refactorings in
the menu confusing. Third, programmers find configuring the refactoring dialog compli-
cated. Configuration dialogs disrupt the programming workflow and impose an overhead
by requiring the programmer to understand the options. Our prior work estimates that
programmers frequently spend up to eight seconds on the dialogs [VCN12]. We term
the second and third problems the invocation and configuration problems respectively
(Section 2.1.2). Indeed, in our own user study, we have observed multiple instances
where programmers struggle with these very problems, confirming their prevalence and severity (Section 2.4.2).

We argue that the invocation and configuration problems stem from the overreliance on menus and dialogs in current refactoring tools. We envision a new approach of extending refactoring tools with a tactile dimension, allowing developers to perform refactorings by directly moving program elements with a drag-and-drop gesture. Using drag-and-drop therefore has two advantages. First, it eliminates the need to navigate through lengthy menus of refactorings. Second, it eliminates the need for a separate configuration step. Through a single movement of selecting the appropriate source and target elements, the programmer is able to both invoke and configure the desired refactoring. Our approach works for all move and extract based refactorings, and our tool supports up to 12 of the 23 refactorings available in Eclipse. These 12 also happen to be some of the most commonly invoked refactoring tools in Eclipse [MHPB09, VCN+12].

Our work makes the following contributions for improving the state of refactoring tools:

1. **Approach**: We introduce a novel refactoring invocation and configuration approach that relies on drag-and-drop of program elements. This technique leverages the **drag source** and the **drop target** of program elements to invoke and configure the refactoring in a single step. The approach is generalizable to different refactorings and different programming languages.

2. **Mappings**: For our approach to work, we needed to define a suitable set of mappings for drag sources and drop targets. To make it more intuitive, we derived the mappings based on the survey responses of 74 participants. Tables 2.1 and 2.2 detail the drag sources and drop targets for the supported refactorings. The mappings serve as useful reference for future researchers and tool developers.

3. **Tool**: We implemented our approach using the mappings in our open source tool, DNDRefactoring, for the Eclipse IDE. DNDRefactoring is supported (i) within a Java editor, or (ii) within and between Package Explorer and Outline views. The Package Explorer and Outline views show a Java element hierarchy tree of the
Java projects and source files. We encourage readers to watch a demo of the tool in action at [onle].

4. Evaluation: We evaluated our tool for its efficiency and usability and answer the following research questions:

RQ1: [Intuitiveness] How intuitive are the drag-and-drop gestures for users?

RQ2: [Efficiency] How efficient is it to invoke and configure drag-and-drop refactoring?

We conducted a within-group controlled user study, where we asked participants to perform non-trivial refactoring tasks using both the existing Eclipse tools and DNDRefactoring. Our results show that DNDRefactoring is not only intuitive but also increases invocation efficiency by decreasing configuration time and error rates compared to traditional refactoring tools, which may in turn invite programmers to use the automated refactoring tools more frequently.

The contributions of Drag-and-Drop Refactoring have been published and presented (by the author of this dissertation) at the 35th International Conference on Software Engineering (ICSE 2013) [LCJ13].

1.3 Tempura: New Temporal Dimension for Code Completion and Navigation

Modern integrated development environments (IDEs) provide automated programming support that make many software development tasks easier. For example, many IDEs offer context-specific programming assist with code completion, providing developers with proposals for completing identifiers, such as type, method, or field names, from a given prefix of element names. IDEs also offer navigation support, allowing developers to quickly find and navigate to type, method, and field declarations. These IDE support features are continuously being studied and improved. For example, Code Recommenders [Ecl] in Eclipse can suggest identifier completion from a given partial name
match and suggest completion for longer code snippets. Others have introduced a tool that automatically synthesizes code snippets using program elements available in the current scope of code [GKKP13]. Some research prototypes additionally use dynamic program information to improve navigation [RHB+12, H10].

While such automated programming support in IDEs help developers’ programming tasks, these features, and thus the IDEs as a whole, operate on one version of the code at a time. Developers working on continuously evolving projects not only have to work with the most current version of code but also frequently need to understand code changes from past revisions made by colleagues or even by themselves. This is because successful software development relies heavily on implicit knowledge, an important subset of which is understanding the history of the code [LVD06]. Developers naturally think about their work on a project in terms of changes made over time. When the developer’s implicit knowledge becomes incorrect or outdated, their productivity is hindered as they are forced to switch context from writing or fixing code to rebuilding the knowledge.

Version control systems (VCSs) build and maintain an explicit knowledge base by recording all changes over the history of a project. VCSs, however, record all the changes to a project’s code base, whether it is a renaming of a method or a spelling correction in comments. Therefore it is left up to developers to sift through all the recorded changes in order to find appropriate changes that impact their programming tasks. In addition, while most modern IDEs provide functionality that supports different VCSs, this functionality mostly exists as add-ons or plugins, for example, EGit and Subversive for Eclipse. There is a distinct separation between the core IDE features like code completion and navigation and VCS features. Current IDEs that restrict core IDE features to operate only on one version of code inherently hamper developers’ productivity, because developers are forced to tediously search through version history in VCSs or manually switch to different revisions when seeking information from past versions of code.

We envision a new approach of extending IDEs by adding a temporal dimension, allowing the familiar programming support in IDEs such as code completion and navigation to work with multiple versions at a time without resorting to manual version switching. Our approach locates types and members from past versions that are relevant to the
current context and presents them through code completion. In addition, our approach also allows developers to search for and navigate to types in any revision, even deleted types. We implemented our approach in an Eclipse plugin called Tempura\(^1\). Tempura currently supports the Java programming language and Git VCS. Tempura provides the temporal dimension to two widely used Eclipse features: code completion and type search navigation. Our work makes the following contributions:

1. **Algorithms**: Two algorithms, namely on-the-fly and indexing, for supporting temporal dimension in IDEs.

2. **Tool**: An open-source prototype Eclipse plugin, Tempura, that embodies our approach for temporal dimension, supporting the Java programming language and Git VCS.

3. **Evaluation**: We evaluate our tool and answer the following two research questions:

   **RQ1**: How efficiently can code history information be collected from a project’s repository? How scalable can the computation be for large real-world projects?

   **RQ2**: Does the history information that Tempura provides through Eclipse’s code completion and navigation features help developers to learn code history more accurately and efficiently?

We conducted an experiment with three large Eclipse projects to show scalability of Tempura (RQ1), and a controlled user study with 10 participants to demonstrate that Tempura allows developers to learn about code changes $17\text{pp}$\(^2\) more accurately and with 50% higher efficiency in terms of rate of information acquisition (RQ2), compared to using EGit that handles code history separately from the current version of code.

All materials, including Tempura’s source code and user study materials are publicly available at http://mir.cs.illinois.edu/tempura.

---

\(^{1}\)Tempura is a Japanese dish, but we derived the name from the word “temporal” for our tool.

\(^{2}\)The term “pp” stands for “percentage point” and is used to represent differences among values that are already represented as percentages.
The initial concept of Tempura has been published and presented (by the author of this dissertation) at the New Ideas and Emerging Results (NIER) track at the 35th International Conference on Software Engineering (ICSE 2013) [LHKM13], and extended contribution is under submission at the 37th international Conference on Software Engineering (ICSE 2015).

1.4 Dissertation Overview

The rest of this dissertation is organized as follows:

**Chapter 2: Drag-and-Drop Refactoring**  This chapter presents Drag-and-Drop Refactoring, a plug-in that extends the refactoring tools in an IDE with a *tactile dimension* to allow developers to invoke and configure refactorings by directly moving program elements instead of via complex GUI-based user interfaces.

**Chapter 3: Tempura**  This chapter presents Tempura, a plug-in that intimately integrate VCS support into the IDE platform by adding a *temporal dimension* to the code completion and navigation functions in an IDE, allowing developers to search for deleted classes, methods, and fields without resorting to a disparate VCS plug-ins.

**Chapter 4: Related Work**  This chapter discusses various work by other researchers that are related to the contributions of this dissertation.

**Chapter 5: Conclusions and Future Work**  This chapter concludes the dissertation and discusses some of the limitations and possible extensions to build upon our work.
Chapter 2

Drag-and-Drop Refactoring: Tactile Dimension for Refactoring Invocation and Configuration

2.1 DNDRefactoring

2.1.1 Motivating Examples

Consider the following scenario. Once a programmer decides on a refactoring to perform, she still has to complete two steps. First, to invoke the tool, she has to navigate through a lengthy and confusing menu (recall that Eclipse, NetBeans and IntelliJ support at least 18 refactorings) and select the appropriate refactoring. She could memorize an elaborate keyboard shortcut but unless it is a refactoring that she frequently uses, she is unlikely to do so (only 1 out of 10 participants in our controlled user study used keyboard shortcuts). Second, to configure it, she has to interact with a dialog containing many detailed options that she might not require and only serve to distract her from her goals (90% of users do not modify the default settings [MHPB09]). Thus, there exists a gap between what she wants to accomplish and how she needs to do it through the current user interface.

To bridge this gap, we allow the programmer to directly manipulate program elements, e.g., variables, expressions, statements, methods, etc. in the IDE, eliminating the need for menus or dialogs. The programmer only needs to identify a program element to serve as the drag source and another program element to serve as the drop target. For instance, to perform an Extract Method refactoring, the programmer would drag the selected expression (source) and drop it into the enclosing class (target) (Figure 2.1a). Similarly, to perform the Move Type to New File refactoring, she would drag the inner class (source) and drop it into the desired package (target) (Figure 2.1b).
2.1.2 Design Rationale

The driving principle behind the design of DNDRefactoring is to streamline the invocation and configuration mechanisms. The current mechanisms, as implemented in modern IDEs, suffer from two problems:

1. **Invocation inconsistencies** – The dominant mechanism of invoking automated refactorings relies on identifying a refactoring by name and selecting it from a lengthy menu. This mechanism has two shortcomings. First, the names are *non-standard*. For instance, Eclipse adheres to Fowler’s naming scheme for Extract Method whereas NetBeans calls it Introduce Method. Second, the grouping of refactorings in the menu is *unpredictable* both within an IDE and across IDEs. For instance, Eclipse places the Rename and Move refactoring in the same category although they are not closely related. Furthermore, while Eclipse groups Extract Superclass together with Pull Up (because they operate on class hierarchies), IntelliJ groups Extract Superclass with the other extract based refactorings and Pull Up in another category. Both these inconsistencies lead to a *hunt-and-peck* style of invoking a refactoring where the programmer has to spend time searching through the menu. This problem was evident in our user study (Section 2.4.2) and also corroborated by Murphy-Hill et al. [MHAB11].

2. **Configuration overload** – The dominant mechanism for configuration relies on dialogs. This is a remnant from the design of the first automated refactoring tool for Smalltalk [RBJ97]. As more complex refactorings were introduced, more complicated configuration options were also made available. However, 90% of refactoring tool users do not modify the default configuration [MHPB09]. Thus, these extra options serve only to confuse and prolong the configuration of refactorings since the user is tempted to read all the options. Moreover, we have evidence from our controlled user study (Section 2.4.2) that some of the options could be erroneously selected by the programmer and could lead to undesired changes to the code.

DNDRefactoring solves both these problems. Because there isn’t a universal naming and grouping of refactorings that everyone can agree upon, we dispense with names altogether: the drag source and drop target determines the refactoring to invoke and
we do not burden the user with remembering names. Similarly, we do not need dialogs because the drag source and drop target already serve as configuration options to the refactoring tool, and we rely on sensible defaults and in-place edit immediately following refactoring where applicable. Our controlled user study suggests that these options are sufficient; the participants are able to complete the tasks without using more complicated configuration options.

Eclipse already provides a workaround for the configuration overload issue with Quick Assist \([\text{only}]\), which performs local refactorings with default values and then allows programmers to make changes. Our implementation of DNDRefactoring in Eclipse leverages the Quick Assist paradigm whenever possible, relying on sensible default configurations.

One could argue that the dialog boxes provide more functionality than just configuration and that eliminating them could be problematic. For instance, the dialog boxes also offer a preview feature that shows the code changes to be performed. However, our prior work \([\text{VCN}^+12]\) report that programmers use the preview feature infrequently and prefer to perform the refactoring and view the code changes directly in the editor. If the user is unsatisfied with the changes, she uses the undo feature to revert the refactoring.

2.1.3 Tool Features

Our implementation of DNDRefactoring in Eclipse allows programmers to invoke existing refactorings by drag-and-dropping program elements (i) within the Java editor or, (ii) within and between the Package Explorer and Outline View. The drag source is the highlighted selection, either a text selection within a Java editor or a tree node in the Package Explorer or Outline View. The drop target is identified by the position of the cursor when the drag source is dropped. For example, within a Java editor, a cursor located in a whitespace anywhere inside a class, but outside any method or field declaration, will identify the target as the class (Figure 2.1a). A refactoring is invoked based on the program element types of the drag source and drop target. If no suitable refactoring is found, the drag-and-drop gesture defaults to textual cut-and-paste.
Table 2.1: Refactorings with Drag-and-Drop: within a Java editor.

<table>
<thead>
<tr>
<th>Drag Source</th>
<th>Drop Target</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local variable</td>
<td>Declaring type</td>
<td>Promote local variable to field (IPE(^1))</td>
</tr>
<tr>
<td>Expression inside method</td>
<td>Same method</td>
<td>Extract temp variable (IPE)</td>
</tr>
<tr>
<td></td>
<td>Between argument brackets of current method signature</td>
<td>Introduce parameter</td>
</tr>
<tr>
<td></td>
<td>Declaring type</td>
<td>Extract method (IPE)</td>
</tr>
<tr>
<td>Statements in method</td>
<td>Declaring type</td>
<td>Extract method (IPE)</td>
</tr>
<tr>
<td>Non-static method</td>
<td>Field variable in declaring type</td>
<td>Move instance method to field type</td>
</tr>
<tr>
<td></td>
<td>Argument type in current method signature</td>
<td>Move instance method to argument type</td>
</tr>
<tr>
<td>Static method of field</td>
<td>Another type in current editor</td>
<td>Move member to target type</td>
</tr>
<tr>
<td></td>
<td>Field variable in declaring type</td>
<td>Move member to field type</td>
</tr>
<tr>
<td></td>
<td>Local variable type in declaring type</td>
<td>Move member to local variable type</td>
</tr>
<tr>
<td>Anonymous class</td>
<td>Declaring type</td>
<td>Convert anonymous to nested type</td>
</tr>
</tbody>
</table>

\(^1\) IPE = In-Place Edit allowed after refactoring is completed.
Table 2.2: Refactorings with Drag-and-Drop: within and between Package Explorer and Outline View.

<table>
<thead>
<tr>
<th>Drag Source</th>
<th>Drop Target</th>
<th>Refactoring</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-static Method</td>
<td>Type of field variable in declaring type</td>
<td>Move instance method to target field type</td>
</tr>
<tr>
<td></td>
<td>Type</td>
<td>Pull-up, Push-down or Move method to target type</td>
</tr>
<tr>
<td>Nested Type</td>
<td>Package</td>
<td>Move nested type to new file + Move type to target package</td>
</tr>
<tr>
<td>Anonymous Type</td>
<td>Type</td>
<td>Convert anonymous to nested type</td>
</tr>
<tr>
<td></td>
<td>Package</td>
<td>Convert anonymous to nested type + Move nested type to new file + Move type to target package</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Pull-up, Push-down or Move field to target type</td>
</tr>
<tr>
<td>Static Members</td>
<td>Another type declared in current editor</td>
<td>Move members to target type</td>
</tr>
<tr>
<td></td>
<td>Type of field variable in declaring type</td>
<td>Move members to target field type</td>
</tr>
<tr>
<td></td>
<td>Type of local variable in declaring type</td>
<td>Move members to local variable type</td>
</tr>
<tr>
<td>Non-static fields</td>
<td>Package</td>
<td>Extract data class + Move type to target package</td>
</tr>
<tr>
<td>Non-static methods</td>
<td>Package</td>
<td>Extract interface</td>
</tr>
<tr>
<td>Static &amp; non-static methods</td>
<td>Package</td>
<td>Extract super class</td>
</tr>
</tbody>
</table>
Tables 2.1 and 2.2 list all the drag-and-drop refactorings that we have implemented for the Eclipse IDE. To the best of our knowledge, the mappings in the tables are new and serve as the first canonical set of drag-and-drop gestures for refactorings. Other mappings for the stated refactorings are possible, but the current mappings were determined based on the survey responses (Section 2.3).

In addition to providing a new method of invocation and configuration, DNDRefactoring also supports two new and useful features that can only be accomplished through drag-and-drop gestures.

1. **Collated refactorings**: A single drag-and-drop gesture can effectively collate several refactorings together. Consider dragging a nested class and dropping it in the current package. This gesture can be translated into Move Type to New File refactoring in Eclipse (Figure 2.1b). What happens if the nested class was dropped in a different package? Naturally, the extended gesture can be interpreted as Move Type to New file refactoring followed by Move type to target package refactoring. This collated refactoring is supported intuitively and effortlessly in a single drag-and-drop gesture using DNDRefactoring. Such a simple collated refactoring is impossible to invoke using the existing invocation and configuration mechanisms in Eclipse. Programmers using the traditional invocation mechanisms are forced to perform two separate refactorings in succession. Collated refactorings are annotated with “+” in Table 2.2.

2. **Precise control**: Another advantage of drag-and-drop is the ability to precisely choose where a drag source is dropped. For example, Extract Method refactoring in Eclipse always creates a new method below the method from which the expression or statements were extracted. However, with DNDRefactoring, programmers’ natural expectation would be to see the extracted method appear exactly where the expression was dropped (Figure 2.1a). DNDRefactoring supports such precise control and allows programmers to decide where to move or extract program elements.
2.1.4 Supporting Floss Refactoring

Murphy-Hill and Black introduced the terms *floss refactoring*, to describe refactorings that occur frequently in small steps that are intermingled with other kinds of program changes, and *root canal refactoring*, which is characterized by infrequent and protracted periods of refactoring. *Floss refactoring* maintains healthy code, and *root canal refactoring* corrects unhealthy code \[MHB08b\]. Studies by Weißgerber et al. \[WD06a\] and Murphy et al. \[MKF06\] suggest that *root canal refactoring* is not practiced often. Murphy-Hill and Black also proposed five principles to characterize a tool that supports floss refactoring. They suggest that such tools should let the programmer:

1. Choose the desired refactoring quickly,
2. Switch seamlessly between program editing and refactoring,
3. View and navigate the program code while using the tool,
4. Avoid providing explicit configuration information, and
5. Access all the other tools normally available in the development environment while using the refactoring tool.

The current refactoring tool in Eclipse violates all five principles \[MHB08b\]. The tools by Murphy-Hill et al. help programmers’ code selection process (i) with syntactic highlights, (ii) by visualizing nested statements as a series of nested boxes, and (iii) with control and data-flow annotations \[MHB08a\]. While the tools helped reduce time and errors during refactoring, they violate Principles 1 and 4 because the tools do not assist programmers with refactoring selection or configuration. The same limitation applies to tools that alert programmers of code smells and opportunities for refactorings \[OSG05\] \[PGN08\]. Murphy-Hill et al. introduced other tools that help with refactoring selection, by mapping directional gestures to refactorings \[MHAB11\]. The tool displays a radial menu with four quadrants, and maps directional gestures (up, down, left or right quadrants) to refactorings. The tool adheres to Principles 1 and 4 because the radial menu displays a more concise set of applicable refactorings and performs the
selected refactoring without requiring explicit configuration from programmers. However, the radial menu is a modal window menu that covers up part of the Java editor and thus violates Principles 2 and 3.

In contrast, we claim that DNDRefactoring satisfies all five principles. DNDRefactoring eliminates the need for programmers to browse through a long list of refactoring menu items and decode refactoring names that aren’t always obvious, therefore Principle 1 is satisfied. In addition, because programmers choose source and target program elements in the editors and views that they are currently working on, Principles 2 and 3 are satisfied. DNDRefactoring also adheres to Principle 4 because it does not interrupt refactoring processes with pop-up prompts, but uses default values to complete the refactoring and then invites programmers to make in-line changes. Lastly, DNDRefactoring does not show modal windows during refactoring, so it also adheres to Principle 5.

2.2 Evaluation Methodology

To measure the utility of DNDRefactoring, we ask and answer the following research questions:

RQ1: [Intuitiveness] How intuitive are the drag-and-drop gestures for users?
Given that there is a large set of possible drag sources and drop targets that can be used to invoke each refactoring, the main challenge is to build a set of mappings that is intuitive to most users. To answer whether drag-and-drop gestures are intuitive, we conducted a survey that asked participants unfamiliar with the drag-and-drop approach to suggest drag-and-drop gestures for 5 randomly selected move and extract based refactorings, and to select refactorings given 5 drag-and-drop gestures. If the majority of users agree on the drag sources and drop targets for each refactoring, it would strongly suggest that there is a set of drag-and-drop gestures that is universally applicable, or intuitive, to all users (Section 2.3).

RQ2: [Efficiency] How efficient is it to invoke and configure drag-and-drop refactoring?
One of the main challenges of automated refactoring tools is the burden of invocation and
Figure 2.1: Drag-and-drop gestures in (a) Java editor for Extract Method refactoring, and (b) Package Explorer for Extract Type to New File refactoring.
Figure 2.2: Examples of Survey Questions
configuration. To answer whether drag-and-drop refactoring is efficient, we implemented DNDRefactoring, an Eclipse plug-in that supports the set of gestures that we determined from RQ1. We then conducted a controlled user study comparing DNDRefactoring to the default Eclipse invocation and configuration mechanisms (baseline). Participants were asked to complete a non-trivial refactoring task using Eclipse with and without DNDRefactoring. We recorded videos of these user study sessions, and analyzed them to measure and compare the time taken to invoke and configure both tools. If the results show that DNDRefactoring is more efficient, then it indicates that DNDRefactoring could be a compelling and complementary addition to the existing tools. In addition, we evaluated and compared the Eclipse and DNDRefactoring interfaces using the Keystroke-Level Model [CNM00], by comparing the number of keyboard and mouse actions required by each interface (Section 2.4).

RQ3: [Usability] How usable is drag-and-drop refactoring?

A tool can be very efficient to invoke and configure, and yet have very little users because of the difficulties involved in using the tool. We wanted to identify the main challenges of using drag-and-drop refactoring compared to the default Eclipse refactoring tools. To answer this question, we (i) asked the participants to provide feedback on DNDRefactoring and (ii) analyzed the videos of the controlled user study that we captured as part of RQ2 to identified obstacles each participant encountered when using both tools. We then coded and merged those obstacles into key categories following the standard data analysis procedure for open-ended survey responses [SC98]. By comparing the categories of obstacles identified in each tool, we can objectively discuss the advantages and disadvantages of each tool and suggest room for improvement (Section 2.4). All survey and study materials, and results are available at [onle].

2.3 Evaluating Intuitiveness

2.3.1 Survey Design

We conducted a survey asking participants to suggest drag-and-drop gestures for refactorings. The survey contained five questions asking participants to suggest a refactoring
Table 2.3: Survey results by category. Each column name corresponds to the particular refactoring asked in the survey.

<table>
<thead>
<tr>
<th></th>
<th>Extract Field</th>
<th>Move Nested to New File</th>
<th>Extract Method</th>
<th>Extract Temp Variable</th>
<th>Introduce Parameter</th>
<th>Convert Anonymous to Nested</th>
<th>Push Down</th>
<th>Pull Up</th>
<th>Move Method</th>
<th>Extract Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Majority</td>
<td>60(81%)</td>
<td>40(54.1%)</td>
<td>46(62.2%)</td>
<td>59(79.7%)</td>
<td>59(79.7%)</td>
<td>55(73.3%)</td>
<td>49(66.2%)</td>
<td>54(73.0%)</td>
<td>51(68.9%)</td>
<td>61(82.4%)</td>
</tr>
<tr>
<td>Alternate</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>1(1.4%)</td>
<td>0(0%)</td>
<td>0(0%)</td>
<td>7(9.5%)</td>
<td>15(20.3%)</td>
<td>8(10.8%)</td>
<td>3(4.1%)</td>
<td>0(0%)</td>
</tr>
<tr>
<td>Infeasible</td>
<td>11(14.9%)</td>
<td>25(33.8%)</td>
<td>19(25.7%)</td>
<td>9(12.2%)</td>
<td>7(9.5%)</td>
<td>5(6.8%)</td>
<td>6(8.1%)</td>
<td>7(9.5%)</td>
<td>1(1.4%)</td>
<td></td>
</tr>
<tr>
<td>Empty</td>
<td>3(4.1%)</td>
<td>9(12.2%)</td>
<td>8(10.8%)</td>
<td>6(8.1%)</td>
<td>8(10.8%)</td>
<td>7(9.5%)</td>
<td>5(6.8%)</td>
<td>6(8.1%)</td>
<td>13(17.6%)</td>
<td>12(16.2%)</td>
</tr>
</tbody>
</table>
given a drag-and-drop gesture, and five questions asking the reverse mapping. Figure 2.2 shows actual samples of the questions asked. Participants were given a 5-minute summary of the study and were asked to complete the survey in 10 minutes.

The survey was conducted in a graduate-level software engineering class. At least 95% of the students have taken a prerequisite course in previous semesters that familiarizes them with Java, Eclipse, and refactoring. All participants were new to the drag-and-drop approach, and the survey was completely voluntary and anonymous.

2.3.2 Results and Observations

We collected 74 survey responses in total. Of those 74 participants, 60 participants (93%) indicated that they have more than 2 years of Java experience, and 58 participants (77%) have more than 2 years of experience with Eclipse. Also, 17 (23.0%), 52 (70.3%), and 5 (6.8%) participants regarded themselves as novice, intermediate, and expert users of the automated refactoring tools in Eclipse, respectively.

We manually coded the responses for each question into two main categories: the majority (the most common response) and minority (Table 2.3). On average 72% of the responses formed the majority. More specifically, on average, 62%, 74% and 90% of refactoring novice, intermediate, and expert users of Eclipse refactoring tool agreed on a mapping, respectively. This result strongly suggests that there is a set of universal drag-and-drop gestures that is applicable for all users. We analyzed all the responses in the majority category and found all of them to be feasible gestures for drag-and-drop. We used these gestures to implement our tool DNDRefactoring.

To better understand the range of responses, we divided the minority category into three sub-categories: alternate, infeasible and empty. The alternate category contains different but reasonable alternative refactorings that could be interpreted from the drag and drop gesture. For example, for a question depicting Extract Method refactoring by dragging a set of statements from inside a method and dropping it just above the method declaration (Figure 2.2a, and Table 2.3), a surveyee answered “[create a] static method

---

1We implemented gestures for 12 refactorings overall. 10 were based on the survey responses, and the remaining two refactorings were conceived by the authors after the survey.
for class Foo". These responses in the alternate group may be supported in future versions of DNDRefactoring. The infeasible category includes responses that either conflict with existing refactorings in Eclipse, are not refactorings, or involve infeasible drag sources and drop targets. Lastly the empty category contains blank responses.

2.4 Evaluating Efficiency and Usability

2.4.1 Controlled User Study Design

We conducted a controlled user study with 10 participants to evaluate the efficiency and usability of DNDRefactoring on several refactoring tasks. Each participant carried out the refactoring tasks twice, once using the default tools in Eclipse and once using DNDRefactoring. The order of the tools was randomized to mitigate the learning effect. Each user study session was recorded in its entirety using either a screencasting software or a video camera. To minimize unfamiliarity with different machines, each participant used their own computer or laptop for the user study.

All 10 participants were computer science graduate students majoring in various sub-disciplines, including software engineering and software testing. All participants had at least 2 years of continuous experience in Java; 6 participants had more than 5 years of Java experience. The majority of participants had from 2 to 5 years of experience in Eclipse. 2, 7, and 1 participants regarded themselves as novice, intermediate and expert users of the Eclipse refactoring tool, respectively. After the user study, each participant was asked to complete a post-study qualitative survey to evaluate their experience with DNDRefactoring. Participation was strictly voluntary with no rewards offered, and invitations to the study was sent through individual emails and departmental mailing lists.

The refactoring tasks given to the participants are based on the Refactoring Lab Session exercise developed at LORE [DVRDB]. The exercise involves multiple refactorings for a Local Area Network simulation program. The individual refactorings are small and independent, thus are more like floss refactoring than root canal refactoring. We made minor modifications to the refactoring tasks in order to remove some duplicated refactorings and include a wider variety of refactorings.
Prior to their individual user study sessions, all participants were given a group tutorial on DNDRefactoring and the official reference on Eclipse’s refactoring tool [onlk]. The DNDRefactoring tutorial showed a short video demonstrating three refactorings, none of which were repeated in the user study. Participants were encouraged to ask questions and to try using both tools on their own code.

We collected data for two metrics: the configurations times (quantitative) and the obstacles encountered (qualitative). All measurements were done post-user-study from the video recordings so as not to affect the participant’s performance on the tasks. If a refactoring was repeated during the user study, only the first execution of the refactoring was considered in evaluation.

For Eclipse’s existing refactoring tool, the configuration time starts from pressing the Refactor menu item (either in the tool bar or the mouse button menu) and ends with pressing the Finish button in pop-up modal windows. For Quick Assist, we started timing from the moment the small options window showing to selecting one option. Lastly for drag-and-drop (for DNDRefactoring or existing simple refactoring support in Eclipse), the time was counted from when the programmer starts her selection to dropping the selection.

We define obstacles as programmers’ actions that are incorrect or unnecessary for invoking desired refactorings, for example, when a programmer selects a wrong refactoring, cancels a refactoring, invokes a refactoring with an irrelevant program element, or when results do not match programmers’ expectations.

2.4.2 Controlled User Study Results and Observations

A. Efficiency

Table 2.4 shows each participant’s configuration times. There was an outlying case where participant #10 introduced a fault when using Eclipse that caused a unit test to fail. She attempted to fix the fault both manually and by using the refactoring tool and thus skewed the results. We felt that while this case may give an insight to the complexity of current refactoring invocation mechanisms in Eclipse, it is not a fair representation of
Table 2.4: Results of the controlled user study\(^1\) - Configuration Time in seconds.

<table>
<thead>
<tr>
<th>PARTIC #</th>
<th>Refactoring</th>
<th>Extract Method</th>
<th>Move Method</th>
<th>Anon. Class to Nested</th>
<th>Move Type to New File</th>
<th>Move Class(^2)</th>
<th>Extract Class</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Eclipse</td>
<td>42.3</td>
<td>48.3</td>
<td>1.1</td>
<td>21.6</td>
<td>6.4</td>
<td>42.1</td>
<td>161.8</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>18.6</td>
<td>12.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>52.2</td>
</tr>
<tr>
<td>#2</td>
<td>Eclipse</td>
<td>106.4</td>
<td>71</td>
<td>40.3</td>
<td>52.9</td>
<td>20.2</td>
<td>32.5</td>
<td>323.3</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>13.8</td>
<td>6.7</td>
<td></td>
<td></td>
<td>22.8</td>
<td>17.6</td>
<td>60.9</td>
</tr>
<tr>
<td>#3</td>
<td>Eclipse</td>
<td>18.6</td>
<td>65.4</td>
<td>13.5</td>
<td>6.2</td>
<td>4.2</td>
<td>151.9</td>
<td>259.8</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>33.5</td>
<td>3.7</td>
<td></td>
<td></td>
<td>16.8</td>
<td>8.9</td>
<td>62.9</td>
</tr>
<tr>
<td>#4</td>
<td>Eclipse</td>
<td>53.3</td>
<td>11.7</td>
<td>33.5</td>
<td>40.4</td>
<td>13.2</td>
<td>40.5</td>
<td>192.6</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>55.9</td>
<td>5.8</td>
<td></td>
<td></td>
<td>39.5</td>
<td>23.5</td>
<td>124.7</td>
</tr>
<tr>
<td>#5</td>
<td>Eclipse</td>
<td>23.7</td>
<td>93</td>
<td>44.1</td>
<td>42.3</td>
<td>9.5</td>
<td>41.5</td>
<td>254.1</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>63.1</td>
<td>5</td>
<td></td>
<td></td>
<td>13.9</td>
<td>11.4</td>
<td>93.4</td>
</tr>
<tr>
<td>#6</td>
<td>Eclipse</td>
<td>10</td>
<td>100.5</td>
<td>50.8</td>
<td>43.5</td>
<td>6.1</td>
<td>24.2</td>
<td>235.1</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>31.3</td>
<td>26.8</td>
<td></td>
<td></td>
<td>15</td>
<td>15.3</td>
<td>88.4</td>
</tr>
<tr>
<td>#7</td>
<td>Eclipse</td>
<td>22.6</td>
<td>46.3</td>
<td>22.3</td>
<td>39.2</td>
<td>7.5</td>
<td>25.1</td>
<td>163</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>22.8</td>
<td>3.4</td>
<td></td>
<td></td>
<td>23</td>
<td>7.6</td>
<td>56.8</td>
</tr>
<tr>
<td>#8</td>
<td>Eclipse</td>
<td>18.8</td>
<td>136.7</td>
<td>28.9</td>
<td>44.8</td>
<td>3.8</td>
<td>23.7</td>
<td>256.7</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>17.6</td>
<td>1</td>
<td></td>
<td></td>
<td>6.8</td>
<td>21.8</td>
<td>47.2</td>
</tr>
<tr>
<td>#9</td>
<td>Eclipse</td>
<td>7</td>
<td>50.7</td>
<td>22.4</td>
<td>5.3</td>
<td>15.3</td>
<td>24</td>
<td>124.7</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>12.6</td>
<td>1.5</td>
<td></td>
<td></td>
<td>13.7</td>
<td>12.9</td>
<td>40.7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average (Median)</th>
<th>Eclipse</th>
<th>DNDR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(22.6)</td>
<td>(22.8)</td>
</tr>
<tr>
<td></td>
<td>(65.4)</td>
<td>(7.4)</td>
</tr>
<tr>
<td></td>
<td>(28.9)</td>
<td>(16.8)</td>
</tr>
<tr>
<td></td>
<td>(40.4)</td>
<td>(14.1)</td>
</tr>
<tr>
<td></td>
<td>(32.5)</td>
<td>(60.9)</td>
</tr>
</tbody>
</table>

\(^1\) Participant #10 introduced a bug while refactoring with Eclipse, thus her data is not included in our analysis.

\(^2\) Time recorded for DNDR is a collated time of Anon. Class to Nested + Move Type to New File + Move Class refactoring.
them. Therefore the data from participant #10 was dropped from our following analysis.

Overall, DNDRefactoring reduced the time spent on configuration by up to 9 times. On average, participants performed the refactorings 3 times faster with DNDRefactoring compared to Eclipse. To validate that this result is statistically significant, we used the Wilcoxon Signed Rank Test (WSRT) to do a pair-wise comparison between the configuration times for Eclipse and DNDRefactoring. We used WSRT to (i) compare the total times for the entire study and (ii) compare the configuration times for each refactoring. WSRT was used instead of the t-test because we cannot assume that the data (configuration time) is normally distributed. Participant #1 did not complete the Extract Class refactoring, so her data was excluded from the calculation of Total Time and Extract Class refactoring. Configuration times for Anonymous Class to Nested Class, Move Type to New File, and Move Class refactorings for Eclipse were collated because the three refactorings can be performed as one refactoring with DNDRefactoring. The p values are reported in the following table; all except Extract Method show statistical significance (p < 0.01).

<table>
<thead>
<tr>
<th>Total Time</th>
<th>Extract Method</th>
<th>Move Methods</th>
<th>Collated Refactorings</th>
<th>Extract Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 0.004</td>
<td>0.715</td>
<td>0.002</td>
<td>0.002</td>
<td>0.004</td>
</tr>
</tbody>
</table>

The results suggest that DNDRefactoring is more efficient compared to Eclipse, except for Extract Method. There are two possible explanations for the inefficiency with Extract Method refactoring. First, the method from which subjects were asked to drag an expression was particularly long, and some found it difficult to drag the expression out of the method while having to scroll the editor. Second, Extract Method is one of the most popular refactorings [MHPB09], and as such, many of the subjects may be familiar and efficient enough with its configuration details.

In addition to configuration time, we also recorded and compared the number of keyboard and mouse actions required by both Eclipse and DNDRefactoring. Programming is inherently mental labor, but not only is the amount of mental effort needed extremely difficult to measure, it can vary greatly between programmers depending on their ex-
Table 2.5: Results of the controlled user study - Programming Effort.

<table>
<thead>
<tr>
<th></th>
<th>Participant #1</th>
<th>#2</th>
<th>#3</th>
<th>#4</th>
<th>#5</th>
<th>#6</th>
<th>#7</th>
<th>#8</th>
<th>#9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard Clicks</td>
<td>Eclipse</td>
<td>445</td>
<td>92</td>
<td>296</td>
<td>84</td>
<td>180</td>
<td>23</td>
<td>82</td>
<td>109</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>320</td>
<td>72</td>
<td>244</td>
<td>151</td>
<td>50</td>
<td>30</td>
<td>40</td>
<td>121</td>
</tr>
<tr>
<td>Mouse Clicks</td>
<td>Eclipse</td>
<td>55</td>
<td>95</td>
<td>112</td>
<td>120</td>
<td>109</td>
<td>103</td>
<td>88</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>33</td>
<td>71</td>
<td>79</td>
<td>122</td>
<td>45</td>
<td>59</td>
<td>50</td>
<td>94</td>
</tr>
<tr>
<td>Mouse Movement (m)</td>
<td>Eclipse</td>
<td>50.80</td>
<td>22.20</td>
<td>8.68</td>
<td>26.04</td>
<td>23.63</td>
<td>12.29</td>
<td>15.02</td>
<td>20.98</td>
</tr>
<tr>
<td></td>
<td>DNDR</td>
<td>35.44</td>
<td>15.00</td>
<td>4.76</td>
<td>25.59</td>
<td>8.92</td>
<td>11.35</td>
<td>10.24</td>
<td>7.71</td>
</tr>
</tbody>
</table>
Programming, however, is also physical labor because programmers write code by typing and interacting with IDEs with a keyboard and a mouse. If programming is purely mental labor, programmers would be able to write code just by thinking. Therefore we concentrated on measuring physical effort required during programming and recorded the keyboard and mouse usage of programmers: keyboard stroke counts, mouse button counts, and the total distance that mouse movements cover.

On average, each participant made 144.7 keyboard strokes when using Eclipse and 118.1 keyboard strokes when using DNDRefactoring. The average number of keyboard strokes saved by using DNDRefactoring was 26.6, (18.4%). Similarly, participants on average clicked the mouse button 101.6 times when using Eclipse and 63.1 times when using DNDRefactoring. The average number of mouse button clicks saved was 38.5 (37.9%). Lastly, participants' average mouse movement covered 19.11 meters when using Eclipse and 12.62 meters when using DNDRefactoring. The average distance saved was 6.49 meters (34.1%). Significant decrease in mouse button clicks and shorter mouse movement with DNDRefactoring, compared to Eclipse, was expected because DNDRefactoring eliminates the need for participants to open either the toolbar or mouse menus for refactoring, for which many programmers use their mouse. Many participants moved their mouse to find a specific refactoring in the menu as well. DNDRefactoring having no modal windows for configuration also resulted in less mouse usage. The more configuration items contained in the modal window means more navigation is required. Even small movements and occasional mouse button clicks can add up if there are a series of refactoring tasks. Relatively smaller decrease in keyboard stroke counts for DNDRefactoring compared to Eclipse was not surprising, as drag-and-drop is exclusively a mouse action. A possible reason for the reduction of keyboard strokes might be because programmers using DNDRefactoring tend to use their mouse when selecting Java elements to refactor, whereas many programmers used keyboards (shift + arrow keys) to make a selection. An interesting trend we noted among almost all participants is that they made changes to their code in uniform ways. For example, if a participant manually changed the access modifier of a method when using Eclipse, she made the same decision when using DNDRefactoring, and vice versa. This suggests that DNDRefactoring does not cause
drastic changes in participants’ programming habits and thus the decreases in programming effort are unbiased.

The error case of participant #10 provided an insightful opportunity to observe how programmers may introduce bugs while interacting with Eclipse’s refactoring interfaces. We were able to retrace and replay her refactoring actions by using Eclipse’s refactoring history and interviewing her after the user study. The bug was introduced while she was moving a method from one class to another, and when one of the references to the moved method was not updated. She invoked the Move refactoring and followed the modal instructions, and opted to view the preview of the changes. Eclipse's refactoring preview window shows a list of Java source files that will be changed by the current refactoring, and allows programmers to exclude any file from the changes. During the interview, participant #10 stated that she remembers seeing one of the files being excluded seemingly by default. Upon replaying her refactoring history we concluded that the exclusion of a file was indeed the source of the bug, but also confirmed that Eclipse by default does not exclude any file from the change list. We conjecture that she had mistakenly or unconsciously excluded a file but because it appeared to her to be a default setting, she accepted it to be correct. While anecdotal, this case demonstrates the danger of configuration overload – it is too easy to erroneously select a wrong option. DNDRefactoring uses the default refactoring configurations and thus streamlines the refactoring process, and does not burden the programmers or provide an opportunity for accidental bugs.

B. Usability

We analyzed the video recordings to identify obstacles that the participants encountered while performing the user study using Eclipse and DNDRefactoring. We iterated through this list to code and merge similar items into categories. Table 2.6 shows the categories we identified. “Cancels” refer to when programmers cancel a refactoring during configuration, or undo an already-executed refactoring. “Manual changes” are when programmers opt to perform any refactoring by hand even though the refactoring tool in use supports it. Other categories names are self-explanatory.
Table 2.6: Results of the controlled user study - Obstacles.

<table>
<thead>
<tr>
<th>Obstacles</th>
<th>PARTICIPANT:</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#1</td>
<td>#2</td>
<td>#3</td>
<td>#4</td>
<td>#5</td>
<td>#6</td>
<td>#7</td>
<td>#8</td>
<td>#9</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>ECLIPSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Manual Changes</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Wrong Refactoring Selected</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Correct Refactoring Unavailable</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Cannot Choose a Refactoring</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Incorrect Configuration</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>7</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>8</td>
<td>3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>DNDRefactoring</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancel</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Manual Changes</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Wrong Source/Target</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Difficulty with Selection</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TOTAL</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

“Correct Refactoring Unavailable” was an unexpected obstacle. The refactoring tool in Eclipse infers to some extent what refactorings a programmer is trying to invoke based on the current cursor position in an editor, and prompts the programmer with a subset of applicable refactorings based on the cursor position. Although useful, this inference can sometimes be counter-intuitive or unexpected, which was the case with participant #3. A slight misplacement of the cursor precluded the refactoring he wanted from appearing in the menu. On the other end of the spectrum was “Cannot Choose a Refactoring” obstacle. A number of participants struggled to pinpoint a desired refactoring in the long list of refactorings.

The participants encountered the most number of obstacles when invoking Move refactoring with Eclipse. Eclipse’s Move refactoring window shows a list of objects with their instance name and type, one from which a programmer can choose to move a method or field to. Many participants found the list confusing, and 6 of them canceled it up to 3 times, often spending much time studying the configuration details.

Many participants also missed a configuration opportunity to change the access modifier when invoking Convert Anonymous Class to Nested Class refactoring, and manually
changed it after the refactoring was completed. In an extreme case, participant #5 opted to perform Extract Class refactoring manually while using Eclipse. Selecting the right program element to invoke refactorings was also difficult with Eclipse. For example, in order to extract a data class with a subset of fields declared in a class, 6 participants selected only the relevant fields in a Java editor and invoked the Extract Class refactoring, but Eclipse by default selects all available fields which *silently* discarded the participants’ preliminary actions. At least one participant did not notice the default configuration and proceeded, eventually undoing the refactoring.

With DNDRefactoring, three participants selected wrong drop targets while invoking Extract Method, Convert Anonymous Class to Nested, and Extract Data Class refactorings. Most notably, a few participants found it difficult to drag an expression out from a long method to invoke the Extract Method refactoring. Also, at least one participant struggled with selecting an expression that is nested within a line of code. Most manual changes made while using DNDRefactoring were for refactorings that DNDRefactoring currently does not support, including Rename and Change Method Signature. On average, DNDRefactoring halved the number of obstacles that participants encountered compared to Eclipse.

2.4.3 Post-Study Qualitative Survey Results

We asked each user study participant to answer a qualitative survey after they completed their tasks. Of the 10 user study participants, 9 found their interaction with DNDRefactoring to be very satisfactory, and 1 found it somewhat satisfactory. Also, 6 participants answered that DNDRefactoring was very comfortable to use while 4 reported that it was somewhat comfortable, and 7 participants found the translation from drag-and-drop to refactorings as expected but 3 found at least one of the refactorings unexpected (refactoring for extracting a data class), or the occasional lack of immediate in-line edit support a little cumbersome. We plan to mitigate these issues in the future, as detailed in the Section ???. All 10 would recommend DNDRefactoring to other people and some also suggested that it should be included as part of the Eclipse IDE. Some participants stated
that DNDRefactoring “[is] very intuitive especially without knowing what the refactoring jargon means” and “saves me the trouble of remembering the exact refactoring to invoke”, and that they “liked that several collated refactorings were invoked with a single action.”

2.5 Limitations

2.5.1 Threats to Validity

A. Internal Validity

We allowed participants to use their own machines for familiarity. These machines varied greatly in terms of specifications and operating systems. Such differences could have affected the configuration time, e.g., using a trackpad instead of a mouse for drag-and-drop, and having a smaller screen requires more scrolling. Also, while we minimize intervention with participants during the controlled user study, the presence of an external viewer (to ensure that we could successfully video capture their session) might subconsciously affect the participants’ performance. Lastly, because participants were aware that DNDRefactoring is a new addition to Eclipse that we have developed, it might have biased them toward/against the approach.

B. External Validity

Our survey and user study participants were advanced undergraduate and graduate students in Computer Science at the University of Illinois. Although collectively the participants have diverse experiences with Java, refactoring, and Eclipse, they might not be representative of all software developers who use refactoring tools. Perhaps within a larger group, different gestures might be suggested for each refactoring. Also, while the refactoring exercise from LORE that we used in our user study is well-known and often used in software engineering classes, it involved only a subset of the refactorings supported by DNDRefactoring. We prioritized keeping the exercise short to enable participants to finish within an hour. Therefore we don’t have data on the performance of DNDRefactor-
ing for the untested refactorings. Lastly, we implemented DNDRefactoring only in Eclipse and compared it to the default refactoring tools in Eclipse. While most refactoring tools in different IDEs follow a similar dialog-based approach, subtle difference between each IDE could still affect the comparison with a drag-and-drop implementation.

2.5.2 C. Limitations of DNDRefactoring

One limitation of DNDRefactoring is the difficulty of translating some refactorings into drag-and-drop gestures. Currently DNDRefactoring only supports move and extract based refactorings. It is difficult, for example, to translate Rename refactorings in drag-and-drop gestures. Second, perhaps mirroring the first limitation, is that some drag-and-drop gestures can be translated into multiple refactorings. For example, drag-and-dropping an expression from within a method to its declaring class can easily translate into both Extract Method and Extract Constant refactorings. In an effort to follow our initial design goal of not interrupting programmers during the execution of refactorings, we default to the Extract Method refactoring. We plan to support multiple refactorings in the future by, for example, prompting programmers with a set of refactoring previews in small tooltips that they can choose from when they drop their drag source.

The tooltip previews can also be used to provide general refactoring previews, as DNDRefactoring currently does not give programmers an option to see a preview of their changes. Recent studies have shown that many programmers in fact do not utilize the preview function in Eclipse [VCN+12]. Tooltip previews, however, may be less interruptive than modal window previews, and can also serve as a visual cue which may help making the DNDRefactoring feature more discoverable.

Some other limitations are specific to our current implementation of DNDRefactoring in Eclipse. First, it is not possible to drag-and-drop AST elements between the Java Editor and Outline View or Package Explorer. Supporting drag-and-drop between these different views can help make some of the drag-and-drop gestures easier to invoke. For example, some user study participants found difficult to drag an expression from inside a long method and drop it outside to perform the Extract Method refactoring. By supporting
Drag-and-drop from the Java Editor to the Outline View, the user can conveniently drop the selected statements to a position in the Outline View without tedious scrolling in the editor.

Drag-and-drop also has some shortcomings. One of the major concerns with drag-and-drop is that the entire gesture has to be completed in a single motion. This can be problematic when the drag source and drop target are obscured in the user interface, e.g., when the users operate on a smaller screens. Suspendable drag-and-drop techniques such as Boomerang alleviate this by allowing the user to first select the drag source, interact with other program elements and resume the drop gesture later [KI07]. Drag-and-drop can also be problematic on larger screens where the mouse has to travel further distances. Pick-and-drop alleviates this by dynamically clustering and displaying the potential drop targets close to the mouse cursor after the source target has been selected [CHBL05]. Many other extensions are possible. Collomb and Hascoët provide a good introduction to other possible extensions and show how they can be unified to support different use cases [CH08]. Future work on DNDRefactoring could incorporate some of these extensions to make it easier to use on smaller or larger screens.
Chapter 3

Tempura: Temporal Dimension for Code Completion and Navigation

3.1 Motivating Examples

Consider a scenario where a field is renamed. Alice, one of many developers working on a project, tries to access a field called NUM_EDGES that she used before by invoking code completion on its declaring class, LexerATNSimulator. Unbeknownst to Alice, however, the NUM_EDGES field was renamed by her colleague. When Alice does not find the NUM_EDGES in the completion proposal list, she suspects that the field is either renamed or removed, which forces Alice to pause her programming task and search through the version history in the project's VCS. One of the biggest challenges Alice faces in her search is the sheer volume of change history, for example, by using Git's log operation, that she has to filter through even before she finds the specific commit that contains the pertinent changes. For example, there may be many commits made by her colleagues since the last time she updated her local Git repository. The declaring class LexerATNSimulator may have undergone many changes. Also, if the commit messages are unclear, or if the commits contain multiple unrelated changes, Alice's task becomes even more complicated and tedious. While many VCS tools provide operations like “blame” that show the last person to make changes to the selected file or line of code, these operations cannot be performed on deleted lines. The search process becomes slower as the size, duration, and number of people involved in a project increase. However, if Alice can still find the NUM_EDGES field in the completion proposal list and use it to pinpoint the exact change that removed the field, she could complete her task much more efficiently.
3.2 Tempura Tool

Tempura embodies our approach of extending IDEs with a temporal dimension by allowing Eclipse to simultaneously operate on previous versions of the code as well as the current version. Tempura collects and indexes all code history information from a project’s VCS in order to provide quick feedback in an interactive use, and supports two main features, (1) temporal code completion, and (2) temporal code navigation with type search. While our Tempura implementation focuses on the Java programming language and the Git VCS, our ideas generalize to other languages and VCSs.

3.2.1 Temporal Code Completion

The Eclipse code completion feature will provide a set of proposals for completing an incomplete expression. Tempura augments this set of proposals with proposals that were possible in any of the previous versions of the code. Figure 3.1 shows the code completion proposals for the LexerATNSImulator class, where historical proposals are displayed in gray. Each historical proposal item also displays pertinent information from the VCS in its tooltip, including the date, author, message, and ID of the commit that removed the particular method or field (Git uses SHA-1 hash for commit ID). The historical code completion proposals cannot be used in the same way as the current code completion proposals, because they will cause compilation errors if inserted into the current code base. Therefore, when a developer selects a historical proposal, Tempura displays a comparison between the revision that last contained the historical proposal and the revision that removed it (diff view, Figure 3.2). It is easy to conjecture that if Alice is using Tempura when searching for the NUM_EDGES field, not only could she very quickly learn that NUM_EDGES was renamed to MAX_DFA_EDGES and assigned a different value in revision 71e0c66, but she could also see other changes that were made in the same commit.

3.2.2 Temporal Code Navigation

Tempura supports temporal code navigation by allowing developers to search for and open any type from any revision using the Open Type in History dialog (Figure 3.3),
Figure 3.1: Tempura’s historical code completion result when invoked on the static LexerATNSimulator type reference. Historical proposals for LexerATNSimulator class are shown in gray, with historical information displayed in tooltip. Example code is from ANTLR4 project.

```java
public abstract class Lexer extends Recognizer<Integer, LexerATNSimulator> implements TokenSource {

    public void pushMode(int m) {
        if (LexerATNSimulator.
```

Declaration changed in commit 71e0ce66 by sharwell on Wed Feb 08 15:15:34 PST 2012

Press Enter to open diff
Commit Message:

Rename NUM_EDGES to MAX_DFA_EDGE, reduce max value to 127
Figure 3.2: Selecting a historical code completion proposal opens Eclipse's diff view, comparing the revision that removed the proposal (left) with the previous revision (right).
including deleted types that are no longer present in the current code base. When a developer searches for a type, the dialog lists all the search matches, where deleted types are shown with a strike-through. For example, Figure 3.3 shows search results for classes whose names start with ParserATN, and shows that ParserATNFactory and ParserATNSimulatorVariationInnerOuterContexts are deleted types that no longer exist in the current code base. Selecting a type from the search result displays the date and the version ID of the commit that last changed the selected type at the bottom of the dialog window. In addition, Tempura identified during parsing that ParserATNSimulator was renamed from a type called v2ParserATNSimulator, and describes the change with an arrow depicting the rename (or move) (Section 3.3.3). If the developer chooses to open a type from the dialog, Tempura opens the type in a *read-only historical editor* (Figure 3.4). The historical editor contains a list of revisions in which the file containing the historical type was modified, along with date, commit ID, author, and commit message. The editor also uses background colors to show changes with respect to a previous revision. For example, Figure 3.4 shows the contents of LexerATNSimulator class in revision 71e0c66, and the line of code highlighted in blue background shows the code that has changed since revision 5225604 (the change corresponds to renaming of the NUM_EDGES field to MAX_DFA_EDGE, shown in Figure 3.2). Similarly, green background highlights added lines of code. In addition, Tempura allows developers to open a diff view comparing the selected revision in the list and its parent revision (Figure 3.2), with the “Show Diff with Previous Revision” button. Tempura also supports navigation within the selected revision. For example, when a developer navigates from one class to another class, Tempura will direct the developer to the destination class in the same revision.

Tempura supports one Git repository at a time, requiring an Eclipse workspace to have projects from a single Git repository. In addition, while a project’s VCS may track different types of files, Tempura handles only source files with the “.java” file extension when processing the VCS.
Figure 3.3: Tempura’s Open Type in History dialog shows historical types, including deleted ones (listed with a strike-through). Selecting a type displays details about the last revision of the type at the bottom of the dialog window. Those types that Tempura identifies to have been renamed (or moved) from another type also describes the change with an arrow.
Figure 3.4: Tempura’s Historical read-only editor with a list of revisions on the left hand side. Blue background color highlights the snippet of code that was changed since the last revision (similarly, green highlights added code).
3.3 Algorithms

The goal of Tempura is to augment Eclipse JDT’s code completion and code navigation operations by providing historical results that were applicable in some past revisions of the given project. Consider the example depicted in Figure 3.1, which shows the result of invoking code completion on a static type reference LexerATNSimulator from the non-static pushMode method in the Lexer type. Also, while it is not identifiable from the figure, note that the LexerATNSimulator and Lexer types do not belong in the same package or share a direct inheritance relationship. Given such a context, Eclipse JDT provides as code completion proposals the members of the LexerATNSimulator type accessible from the pushMode method of the Lexer type, namely public static members (those displayed in colors). Tempura then augments the set of proposals with old public static members of LexerATNSimulator type that no longer exist in the current revision of the type.

More precisely, we define code completion and navigation operations of Eclipse JDT and Tempura as follows:

- For Eclipse JDT’s code completion, let us define the Java element on which code completion is invoked as the receiver element $E$. $E$ can be a variable or a static type reference, on which code completion was invoked using the dot operator. In such a case, the type of $E$ can be resolved. $E$ can also be a string token for prefix of an element name, on which code completion was invoked without the dot operator, in which case the type of $E$ cannot be resolved. We call the type of $E$, either resolved or unresolved, the receiver type. Let $c$ be the element from which the code completion was invoked. We call $c$ the caller element, which can be a method or a type, and the enclosing or resolved type of $c$ the caller type. receiver element $E$ and caller element $c$, as well as the package and inheritance relationships between the receiver type and caller type, form the context of the code completion invocation that determines which program elements (types, methods, or fields) are accessible via the code completion result. Lastly, let $n$ be the revision of the project currently
open in Eclipse.

For example, in figure 3.1, the static type reference LexerATNSimulator is the receiver element \( E \). And because the code completion is invoked with the dot operator, the receiver type can be resolved, which in this case is again LexerATNSimulator. The caller element in figure 3.1 is the void pushMode(int m) method, and its enclosing type, Lexer, is the caller type.

With \( E \), \( c \), and \( n \) defined, let \( E_{cn} \) be the set of proposals that Eclipse JDT computes when code completion is invoked on the receiver element \( E \) from the caller element \( c \) in revision \( n \) of the project. Similarly for Eclipse JDT’s code navigation with type search, let us define that the current revision of project \( P_n \) is a set of types. Then the type search result \( S \) for a prefix \( p \) is a set of types in the project \( P \) whose names start with \( p \): \( S_n = \{ T | T \in P_n \land T \text{’}s \text{ simple or fully-qualified name begins with } p \} \). Note that temporal code navigation with type search, unlike temporal code navigation, is not restricted to any context.

- Given the definitions of Eclipse JDT’s code completion and navigation operations, the temporal code completion proposals that Tempura computes is a union of \( E_{cn} \) where \( r \) ranges from some revision \((r = m)\), the first revision in which both the receiver type and caller type exist in the project, to the second most current revision \((r = n - 1)\) of the project, less the proposals in the set \( E_{cn} \); \( \bigcup_{r=m...n-1} E_{cr} - E_{cn} \).

  We subtract the set \( E_{cn} \) so that Tempura does not duplicate the proposals computed by Eclipse JDT. Similarly, for temporal code navigation with type search, the search result that temporal code navigation computes is a union of \( R_r \) where \( r \) ranges from first revision \((r = 0)\) to the current revision \((r = n)\) of the project; \( \bigcup_{r=0...n} S_r \).

- Note that we restrict the definitions of the sets \( E_{cn} \) and \( E_{cr} \) to contain only API-level program elements, that is, program elements at member granularity (methods and fields), and disregard local variables that Eclipse JDT’s code completion can propose when applicable.
Based on these definitions, we implemented two algorithms that build the union set \( \bigcup_{r=m...n-1} E_{cr} - E_{cn} \) for temporal code completion and/or the union set \( \bigcup_{r=0...n} R_r \) for temporal code navigation, namely (1) checkout algorithm, and (2) indexing algorithm.

### 3.3.1 Checkout Algorithm

The most straightforward way of building the union set \( \bigcup_{r=m...n-1} E_{cr} - E_{cn} \) for temporal code completion is to check out every revision of a project in which both the receiver type and caller type exist \((r = m...n-1)\) and invoke code completion on element \( E \) from caller element \( c \) in each revision. However, this approach proved to be prohibitively expensive as our initial attempts took roughly 30 seconds to check out and build one revision of ANTLR4 project. Even though ANTLR4 would be considered a small project with total of 1636 revisions at the time of experiment, it would still have taken over 13 hours to check out and build each and every revision. As such, checkout algorithm mimics checking out every revision of the entire project, by only checking out every revision of the files containing the receiver type and its superclass and interfaces. This algorithm builds the union set \( \bigcup_{r=m...n-1} E_{cr} - E_{cn} \) by replacing the content of these files with their past versions in chronological order and using Eclipse JDT’s code completion engine before restoring the files’ contents.

While checkout algorithm provides a lightweight solution for Tempura, it has a number of limitations. First, it only supports code completion and not type search navigation. More precisely, the computation for code completion does not lend itself to support type search for deleted types, unlike indexing algorithm (Section 3.3.2). Similarly, if the receiver type at some point in the past extended a superclass that is since removed and no longer exists in the current code base, checkout algorithm will not collect proposals from the superclass. Therefore checkout algorithm exposes only the history of types that still exist in the code base. Second, the runtime computation becomes progressively inefficient as the receiver type’s history increases, rendering it unusable for types with many revisions (Table 3.2). This issue can be addressed to some extent by limiting the number of revisions from which Tempura collects historical proposals, for example, from
the last \( m \) revisions or from a developer's last commit. Lastly, checkout algorithm cannot collect historical proposals for types that are defined in the same file as the type from which code completion is invoked, including code completion on \texttt{this} keyword. This is because Eclipse JDT uses the cursor position in the file to compute receiver types, and temporarily replacing the content of a file potentially changes the identifiable element at the cursor position.

We believe that checkout algorithm is functional for projects with short history, but we also believe that a more rigorous support was possible, which resulted in the indexing algorithm that addresses the shortcomings of checkout algorithm, described next.

3.3.2 Indexing Algorithm

The main goal of this algorithm was to achieve usable runtime efficiency and to support both temporal code completion and navigation. Consider again the code completion example shown in Figure 3.1. The proposals that Eclipse JDT provides (those displayed in colors) are a subset of all members of the \texttt{LexerATNSimulator} type (receiver type) that are accessible from the \texttt{Lexer} type (caller type) via the static type reference. Then the proposals that Tempura provides (those displayed in gray) are a subset of all members of any past revision of \texttt{LexerATNSimulator} type. Similarly, Eclipse JDT's type search result for a given prefix of element name is a subset of all types declared in the project, then Tempura's type search result for the prefix is a subset of all types that were declared in any old revision of the project.

More precisely, we make the following observations:

- For Eclipse JDT's code completion, if receiver type can be resolved, the set \( E_{cn} \) is a subset of all members of the current revision of the receiver type (and its superclass and interfaces). If receiver type cannot be resolved, \( E_{cn} \) is a union of a subset of all types in the current revision of the project and a subset of members of the current revision of the caller type. Similarly, it is obvious from the definition of Eclipse JDT's code navigation operation with type search that the set \( S_n \) is a subset of all
types in the current revision of the project.

- Given the first observation, it trivially extends that the set $\bigcup_{r=m...n-1} E_{cr} - E_{cn}$ is either a subset of all members of any past revision of the receiver type, or a union of a subset of all types in any past revision of the project and a subset of all members of any past revision of the caller type. Also, the set $\bigcup_{r=0...n} S_r$ is a subset of all types in any past revision of the project.

- Based on the second observation, a set of all types and their program elements found in any revision of the project is a superset of $\bigcup_{r=m...n-1} E_{cr} - E_{cn}$. In addition, the superset of $\bigcup_{r=m...n-1} E_{cr} - E_{cn}$ includes the similar superset of $\bigcup_{r=0...n} S_r$.

Indexing algorithm is based on these observations, and it implements a two-step procedure. First, it processes a project’s repository to index types and their program elements from past revisions of the project, building the superset of $\bigcup_{r=m...n-1} E_{cr} - E_{cn}$. Second, it filters code completion and type search results from the index and presents them to developers.

1. **Indexing Program Elements from Past Revisions of Project:** This step takes a Git repository of a project as an input and produces two indices for program elements, namely Type Index and Member Index. These indices are constructed only once by processing the repository when Tempura is first initialized, and they are persisted on disk. Tempura processes subsequent commits as they take place, updating the indices each time. Each program element is indexed in a simple data object format that records string values of the corresponding element’s syntactic components, as well as other information such as the revision ID and file ID which are necessary for Tempura’s operations. The Type Index maps a type's fully-qualified name to a data object representing the type, and the Member Index maps a type's fully-qualified name to a set of data objects representing members declared in the type. Figure 3.5 describes the type hierarchy of the data objects that Tempura indexes, where the Type Index stores data objects of AbstractHistoryType type, and the Member Index stores the data objects...
of AbstractHistoryMember type. AbstractHistoryType records extra information for handling context changes and change inferences, details of which are described in section 3.3.3.

Given a project’s Git repository, Tempura traverses the commit log of the repository in chronological order to process the changes made to the project over time. Git records the changes in each commit as snapshots of files that are added, modified, deleted, or renamed to and from the previous version [Cha09]. The rename changes are simply pairs of deleted and added files formed by using Git’s rename detection capability, and Section 3.3.3 describes how Tempura uses it to identify renamed or moved types. Tempura parses the file snapshots in each commit using Eclipse’s Java parser when processing a commit in order to extract program elements from each file, i.e., declarations of types defined in a file, and their method, field, and inner class declarations (and their members recursively). Tempura disregards deleted files in a commit as the files would have been parsed when it was either added or modified before being deleted. Figure 3.6 shows the pseudo code for parsing the file snapshots.

Parsing a Java file every time it is added, modified, renamed, or deleted, effectively records for each program element declared in the file the revision in which it was last observed in the file. For example, NUM_EDGES field was last observed in the LexerATNSimulator class in a file in revision 5225604. The file’s next revision, revision 71e0c66, then renamed the field NUM_EDGES to MAX_DFA_EDGE, effectively removing the field from the class. Therefore, by also recording the next revision’s ID (childCommitID field in HistoryElement type, Figure 3.5), Tempura can easily and quickly show a diff view when a historical proposal for a type or member is selected.

Figure 3.7 depicts an overview of how Tempura processes each commit by parsing the file snapshots and extracting program elements. The LexerATNSimulator type is converted into a data object, where its modifiers, name, and package name are stored as string values, “public”, “LexerATNSimulator”, and “org.antrl.v4.runtime.atn”, respectively, among other information. This data object is indexed in the Type Index, with the type’s fully qualified name as the key. Similarly, the MAX_DFA_EDGE field and the toInt method extracted from parsing the LexerATNSimulator type are converted into data ob-
Figure 3.5: Type hierarchy of HistoryElement, a data object for storing program elements extracted from history. Each of these data object records the corresponding element's syntactic components.

Legend:
1. The ID of the commit (SHA of the commit object) in which the file containing the type or member was modified.
2. ID of the file's next commit.
3. ID of the file (SHA of the blob object)
Input:
Repository Repo

Output:
  types: Map < String, AbstractHistoryType >
  members: Map < String, Set < AbstractHistoryMember > >

1: for each Commit C in Repo do  
2:   for each file F in C do  
3:     switch (F)  
4:       case F is added in C:  
5:         ast ← abstract syntax tree parsed from F  
6:       case F is modified in C:  
7:         ast ← abstract syntax tree parsed from post-modification version of F  
8:       case F is renamed in C:  
9:         ast ← abstract syntax tree parsed from post-rename version of F  
10:     end switch  
11:   end for  
12: end for  
13: for each node in ast do  
14:   switch (node)  
15:     case node is a top-level class, interface, or enum declaration:  
16:       types ∪ = <FQN of T → TopLevelHistoryType representing node>  
17:     case node is nested class, interface, or enum declaration:  
18:       types ∪ = <FQN of T → NestedHistoryType representing node>  
19:     case node is a method declaration:  
20:       T ← declaring type of node  
21:       members.get(FQN of T) ∪ = HistoryMethod representing node  
22:     case node is a field declaration:  
23:       T ← declaring type of node  
24:       members.get(FQN of T) ∪ = HistoryField representing node  
25:   end switch  
26: end for  
27: return types and members
Figure 3.7: Processing of commits in Indexing Algorithm. It traverses the commits in chronological order, parsing file snapshots in each commit to extract program elements. The extracted program elements are stored as simple data objects indexed by the (enclosing) type’s fully-qualified name.
jects and are indexed in the Member Index. The Member Index also shows a data object for `NUM_EDGES` that was parsed when Tempura processed the commit 5225604 which had a snapshot of the modified file containing the `LexerATNSimulator` type.

Note that Tempura stores only a single record for each distinct program element. That is, a `HistoryElement` object does not keep track of all the revisions in which the element was observed, but only the last revision, because recording only the last revision provides sufficient information to support historical code completion and navigation. However, keeping track of all revisions may also provide beneficial information for developers, for example, the first and last revisions in which an element was observed can be used to indicate the life of the element, and it is a possible extension for the future version of Tempura.

2. Filtering Temporal Code Completion and Navigation results: The Type Index and Member Index persisted on disk during the indexing phase are read into the memory when Tempura is activated, and Tempura searches the indices when code completion or type search is invoked. When a developer invokes code completion on a receiver element $E$ from a caller element $c$, Tempura uses Eclipse JDT to identify the program elements necessary to rebuild the context; receiver element $E$, caller element $c$, receiver type, and caller type. It may also include a string token if a developer used a prefix of a name to narrow the desired code completion proposal or type search results. Figure 3.8 shows the pseudo code for filtering the indexed data objects when code completion was invoked on a receiver element $E$ from a caller element $c$.

As stated earlier, if code completion was invoked on a variable or a static type reference using a dot operator, the receiver type can be resolved. In such a case, Tempura needs to provide as proposals the old members of the receiver type. Tempura does so by using the receiver type's fully-qualified name as the key to retrieve a set of data objects from the Member Index. If code completion was invoked on an element name prefix without the dot operator, the receiver type cannot be resolved, in which case Tempura needs to provide as proposals the old types whose names start with the prefix, and the old members of the caller type whose names start with the prefix. Therefore Tempura
### Figure 3.8: Indexing algorithm - Filtering

**Input:**
- `targetElement`: type of the reference on which code completion is invoked, static type, variable, or undefined
- `targetType`: resolved type for which proposals are to be collected
- `callerElement`: resolved call site element, method, block, or type
- `callerType`: resolved type containing the code completion call site
- `prefix`: prefix of element name entered by developer, could be empty

**Output:**
- `elements`: Set `< HistoryElement >`

```
1: elements ← Set `< HistoryElement >
2: typeIndex ← type index from parsing phase of the algorithm
3: memberIndex ← member index from parsing phase of the algorithm

4: if targetType = undefined then  \(\triangleright \) code completion invoked on an empty/partial prefix
5:     elements \(\cup\) = all objects from typeIndex whose key begins with prefix
6:     if callerElement is a static member then
7:         elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(callerType's FQN) ∧
8:                        o.modifier includes "static" ∧ o.name begins with prefix \}
9:     else
10:        elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(callerType's FQN) ∧
11:                        o.name begins with prefix \}
12: end if
13: else  \(\triangleright \) code completion invoked on a reference, targetType can be resolved
14:     elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(targetType's FQN) ∧
15:                        o.modifier includes "public" ∧
16:                        o.modifier excludes "abstract" ∧ o.name begins with prefix \}
17:     if targetType = static type reference then
18:         elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(targetType's FQN) ∧
19:                        o.modifier includes "static" ∧ o.name begins with prefix \}
20:     else
21:         if targetType = callerType then
22:             elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(targetType's FQN) ∧
23:                         o.modifier includes "private" ∧ o.name begins with prefix \}
24:         end if
25:     end if
26:     if targetType and callerType are in the same package then
27:         elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(targetType's FQN) ∧
28:                         o.modifier excludes "public", "private", "protected" ∧
29:                         o.name begins with prefix \}
30:     end if
31: end if
32: if targetType is a supertype of callerType then
33:     elements \(\cup\) = \{ ∀o ∣ o ∈ memberIndex.get(targetType's FQN) ∧
34:                         o.modifier includes "protected" ∧ o.name begins with prefix \}
35: end if
36: end if
37: return elements
```
retrieves data objects from the Type Index whose key begins with the prefix, and also uses the caller type’s fully-qualified name as the key to retrieve the data objects and filters them to find those whose name begins with the prefix. Tempura then further filters the retrieved data objects using the prefix and following the Java language’s accessibility semantics. In addition, Tempura discards retrieved data objects whose commitIDs are of the ancestor commits of \( m \), the first commit in which both the receiver type and caller type existed. This is to prevent displaying a proposal that was not usable even in the past, for example, a method of the receiver type was removed before the caller type was created. For type search, Tempura matches the prefix to the fully-qualified or simple names of all the types indexed in the Type Index. Unlike temporal code completion, however, temporal type search also includes the types that are present in the current version of code, along with deleted types, because the read-only historical editor allows developers to choose any version of the type using the list of revisions on the left-hand side (Figure 3.4).

Limiting runtime computations to simple map lookup and filtering for code completion and type search allows indexing algorithm to provide fast response time regardless of the length of project or receiver type’s history (Table 3.2).

3.3.3 Challenges

There are a number of aspects in how Tempura handles code history via code completion and navigation operations that posed important and interesting challenges. We describe three main challenges and how Tempura addresses them:

1. **Handling Changes that Affect Temporal Code Completion Results**: As described earlier, Tempura collects code completion proposals from all past revisions in the given invocation context, which are a set of receiver and caller elements and types that determine which program elements to include in the proposals list. Those program elements, however, could also have had complicated history, which can affect the temporal code completion results. While such changes would have little to no impact on temporal
code navigation with type search, temporal code completion is more sensitive to them. We have identified three such case and their (partial) solutions.

- Consider a type \( T \) that extends a super type \( T' \) in the current revision. When a developer invokes code completion on an element \( E \) whose type is \( T \), the proposals include some members from \( T' \) that are accessible through \( T \). However, it is possible that \( T \) extended a different super type in the past, \( T'' \), in which case Tempura's historical code completion also needs to include accessible members of \( T'' \), or otherwise Tempura would ignore some parts of the history. Tempura handles the possible changes in inheritance relationship by recording a type's super type and interfaces during indexing (interfaces and superclasses fields in AbstractHistoryType type, Figure 3.5). During filtering, Tempura recursively searches a type's all past and current super types and interfaces to collect accessible fields and methods. Filtering, however, would need to be improved in the future to take into account the possible changes in the past super types and interfaces (and theirs, recursively). Any changes, or more specifically deletion of members, of \( T'' \) after it was unextended by \( T \) should not affect the temporal code completion results for \( E \). This could be implemented by recording the last revision in which \( T'' \) was extended and filtering out members that existed only in the subsequent revisions.

- Changes in non-identifying components of an element can also affect the result of temporal code completion. For example, a type is identified by its fully-qualified name, regardless of the value of its access modifier. However, changes in the access modifier can change whether or not the type is included in some temporal code completion results. Similarly, identifying members of a type just by comparing their signatures during indexing presents some limitations. For example, while access modifiers and return types of methods are not part of their signatures, any change in them affects the resulting set of code completion proposals. Tempura therefore includes such components when identifying the data objects in both the Type Index and the Member index during indexing. For example, if a field \( F \) of a type \( T \) had its access modifier changed from protected to public at some point in
the past, the set of data objects in the Member Index with \( T \)'s fully-qualified name as the key will have two data objects representing the field \( F \), one with protected access modifier and the other with public access modifier.

- Lastly, there may be cases where seemingly identical members in different revisions may in fact be different. For example, consider a type \( T \) that had a method with the following signature \( \text{setLocalTime}(	ext{LocalTime} \ t) \) in revision \( r1 \). In a later revision \( r2 \), the type \( T \) was modified where an import statement \( \text{import java.time.LocalTime;} \) was changed to \( \text{import org.joda.time.LocalTime;} \). While the change in import statement clearly changes the signature of \( \text{setLocalTime} \) method, simple parsing cannot identify the change of the argument \( \text{LocalTime} \)'s type. Tempura therefore computes simple type resolution whenever possible to identify actual types by searching in import statements for their simple names. If an import statement contains a wildcard ("*"), Tempura only stores the simple name of the types. If the simple name is not found in the import statements or the \text{java.lang} package, and there is no wildcard import statement, then Tempura uses the declaring type's package name to resolve the types.

2. **Supporting Branches:** With distributed VCSs such as Git, the code history of a large project is rarely linear. Any project of respectable scale will most likely have multiple branches. Branches present interesting challenges when merging, for example, some researchers aim to predict merge conflicts ahead of time by identifying code changes in branches that relate to code changes in the main development branch [TZC11]. Multiple branches and their merging also pose an important issue for temporal code completion and navigation, as presenting in the current branch a code completion proposal or type search result that exists only in a different branch can confuse developers and lead them to build a wrong implicit knowledge of their code.

A branch in Git is a lightweight movable pointer to one of the commits. When a repository is created, Git provides a default pointer called \textit{master}. When a new branch is created, for example, called \textit{test}, Git simply creates a new pointer called \textit{test}. These
pointers point to the latest commit made in their corresponding branches. Git also provides a special pointer called **HEAD** that points to one of the pointers to mark the current branch. For example, if a developer currently on the **master** branch switches to the **test** branch, the **HEAD** branch that was pointing to the **master** pointer moves to point to the **test** pointer. When the developer makes a new commit in the **test** branch, the **test** pointer moves forward along with the **HEAD** pointer [Cha09]. The path from the commit that a branch pointer points to (i.e., the latest commit in the branch) to the initial commit forms the branch’s commit log.

In order to build the correct indices for each branch, Tempura maintains a pair of the Type Index and Member Index for each branch. As Tempura processes the commits in chronological order, it checks if a commit is in the log of each branch, and updates a branch’s indices pair only if it is. Separate pairs of indices for each branch is necessary because the indices are effectively compressing the history by processing commits in chronological order, and the timestamp on each commit conveys no information about which branch it belongs to.

Diverged branches of a project’s repository may merge again as the project evolves, creating a merge commit. Handling the merge commits presents challenges for Tempura during indexing, most importantly because a merge commit effectively groups and duplicates the changes that were made in individual branches. For example, when Tempura processes each commit in the repository in chronological order, an addition of a field in one of the branches prior to merging will appear to be added again in the merge commit, resulting in an inaccurate commit information being indexed with the field. This is because non-conflicting merges essentially duplicate the changes made in separate branches. This means, in a larger scale, that the entire history of a project will be represented by few merge commits. Our solution is therefore to skip the merge commits during indexing. However, this is only a partial solution for non-conflicting merge commits. If a merge required developers to manually resolve conflicts, which may involve removal or addition of members, excluding merge commits may lose valuable information. One possible solution is to parse the file snapshots in merge commits only if it is a conflicting merge. However, while identifying a future merge commit as either con-
Figure 3.9: Code completion results when invoked on a different new branch in ANTLR4 repository, created following the revision shown in Figure 3.1. Several commits were made in the new branch, during which two static final integer fields, namely TEST and TEST_2, were added and then removed in different revisions. Compared to the code completion results shown in Figure 3.1, Tempura shows the two extra history proposals that pertains only to the new branch.
flicting or non-conflicting would be trivial, identifying for a past merge commit would require re-merging of the involved branches to determine the conflict status.

3. Inferring Changes: While inferring changes, specifically refactorings [KGLR10, DCMJ06, NVC+12, PRSK10, KNG07], is out of scope of our work, Tempura infers class rename and move refactorings by leveraging Git's rename/move detection capability. Git can easily detect renaming of a file with no changes in its content because Git tracks file contents and not file names. However, because the Java syntax requires changes in both the file name and class (or package) name in the file content in case of a class rename (or move), Tempura uses Git's rename detection threshold score to infer class rename or move refactorings. The threshold score is the minimum content similarity in percentage required to pair a deleted and an added files in a commit as a rename (or move). Tempura sets the threshold to a conservative 99, which means that only deleted and added files whose contents in bytes are at least 99% identical are paired up as a renamed (or moved) file. The high threshold value, while selected arbitrarily, takes into account some cases where a file contains proportionately less program elements than non-program elements such as comments, for example, a simple interface with long copyright header comments. When indexing, Tempura keeps a record of pairs of paths indicating pre- and post-rename (or move) files, and uses the path pairs to identify the fully-qualified names of pre- and post-rename (or move) classes (including non-public and inner classes declared in a file). These fully-qualified names are recorded in a HistoryElement to indicate the class from/to which a class was renamed (or moved), and this information is displayed to developers in the Open Type in History dialog (Figure 3.3). Because the rename/move detection is based on Git's byte comparison and not on systematic analyses of the Java programs, Tempura makes conservative heuristic decisions when required. For example, if Git detects that a file containing one class is renamed to a new file containing multiple classes, Tempura chooses not to report the rename change. Similarly, if Git detects that a file containing $n$ classes is renamed to a new file containing the same number of classes, Tempura checks the equality of each class’s simple name before and after the change to report only the moved classes.
Tempura limits change inference only to classes because inferring changes on members, or more precisely inferring refactorings, is a non-trivial problem and an active research topic. For example, researchers have extracted refactorings from software archive to help detect possible sources of errors and capture intent of changes [WD06b], and proposed a heuristic-based algorithm that detects renamed methods between two versions of code [KPG05]. Tempura may be extended to leverage existing research tools to infer refactorings in the future. For example, Negara et al.’s approach of assigning unique IDs to every AST node when tracking changes suggests a promising direction for Tempura.

3.4 Evaluation

We evaluated Tempura in two ways. First, we evaluated Tempura’s efficiency in indexing historical data from a project’s repository and runtime computation. Second, we conducted a controlled user study to compare and evaluate Tempura against EGit [egi], a widely used Git plugin for Eclipse, in helping developers learn about code history.

Through both evaluations, we answer the following research questions:

RQ1: How efficiently can code history information be collected from a project’s repository? How scalable can the computation be for large real-world projects?

RQ2: Does the history information that Tempura provided through Eclipse’s code completion and navigation features help developers to learn code history more accurately and efficiently?

3.4.1 Indexing and Runtime Efficiency

In order to answer RQ1, we evaluated Tempura’s efficiency in collecting API information from Git repositories of three large-scale projects. The experiment was performed on a dual-core 2.66 GHz MacBook Pro, with Eclipse 3.8 and Java 1.6. The results are shown in Table 3.1. The values for the Time (s) column were calculated by averaging three separate indexing processes for each project. # of Files Parsed column shows the total number of files that Tempura parsed. We also included ANTLR4 project, and the LANSimulation project was used in our user study, described in Section 3.4.2. The indexing takes place
### Table 3.1: Indexing algorithm's indexing of every Java file in each revision from Git repositories.

<table>
<thead>
<tr>
<th>Project</th>
<th># of Commits</th>
<th>Time (s)</th>
<th># of Files_parsed</th>
<th>Parsed Data Size (bytes)</th>
<th>File Size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.eclipse.jdt.ui</td>
<td>26684</td>
<td>308.732</td>
<td>118225</td>
<td>1486824832</td>
<td>41287550</td>
</tr>
<tr>
<td>org.eclipse.jdt.core</td>
<td>21165</td>
<td>711.921</td>
<td>83194</td>
<td>4661719552</td>
<td>24077726</td>
</tr>
<tr>
<td>org.eclipse.platform.ui</td>
<td>25052</td>
<td>237.2875</td>
<td>102567</td>
<td>1259510656</td>
<td>42480222</td>
</tr>
<tr>
<td>ANTLR4</td>
<td>1636</td>
<td>28.202</td>
<td>7781</td>
<td>108760168</td>
<td>4979603</td>
</tr>
<tr>
<td>LANSimulation</td>
<td>21</td>
<td>0.962</td>
<td>54</td>
<td>416672</td>
<td>15813</td>
</tr>
</tbody>
</table>

### Table 3.2: Temporal Code Completion Invocations for both algorithms. On-the-fly algorithm failed to compute code completion for top three classes due to their long history.

<table>
<thead>
<tr>
<th>Class</th>
<th># Revisions</th>
<th># Hist. Proposals</th>
<th>Time (s)</th>
<th># Hist. Proposal</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.eclipse.jdt.core.JavaCore</td>
<td>712</td>
<td>-</td>
<td>-</td>
<td>599</td>
<td>2.08</td>
</tr>
<tr>
<td>org.eclipse.jdt.internal.compiler.problem.ProblemReporter</td>
<td>611</td>
<td>-</td>
<td>-</td>
<td>1385</td>
<td>1.30</td>
</tr>
<tr>
<td>org.eclipse.jdt.internal.compiler.parser.Parser</td>
<td>556</td>
<td>-</td>
<td>-</td>
<td>1699</td>
<td>1.36</td>
</tr>
<tr>
<td>org.eclipse.jdt.internal.compiler.lookup.ReferenceBinding</td>
<td>212</td>
<td>13510</td>
<td>34.07</td>
<td>1229</td>
<td>2.58</td>
</tr>
<tr>
<td>org.eclipse.jdt.internal.compiler.Compiler</td>
<td>125</td>
<td>4623</td>
<td>18.48</td>
<td>58</td>
<td>0.57</td>
</tr>
<tr>
<td>org.antlr.v4.runtime.atn.LexerATNSimulator</td>
<td>107</td>
<td>1608</td>
<td>8.51</td>
<td>200</td>
<td>0.58</td>
</tr>
</tbody>
</table>
when Tempura is first installed, and subsequent revisions are parsed immediately following a commit to the repository, and thus incur only negligible cost. We believe the results demonstrate the scalability of Tempura’s code history information collection even for very large and dynamic projects.

In addition, we also evaluated Tempura’s runtime efficiency by invoking code completion on six classes semi-randomly selected from the org.eclipse.jdt.core project (Table 3.2). The top three classes, namely JavaCore, ProblemReporter, and Parser classes, are defined in files that have undergone the most number of revisions in the project. Other classes from the project were randomly selected, and we also include the LexerATNSimulator class from the ANTLR4 project that we use as an example. The # Hist. Proposals column indicates the number of proposal candidates each algorithm inspects in order to collect the historical proposals, and the Time (s) shows how long it takes for each algorithm to collect historical proposals, averaged over three invocations. Without any restrictions on the number of revisions to inspect, on-the-fly algorithm failed to collect historical proposals from the top three classes due to their long history. Indexing algorithm, on the other hand, shows fast response times.

<table>
<thead>
<tr>
<th></th>
<th># Hist. Proposals</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaCore</td>
<td>50</td>
<td>5.2</td>
</tr>
<tr>
<td>ProblemReporter</td>
<td>100</td>
<td>10.4</td>
</tr>
<tr>
<td>Parser</td>
<td>150</td>
<td>15.6</td>
</tr>
<tr>
<td>LexerATNSimulator</td>
<td>200</td>
<td>20.8</td>
</tr>
</tbody>
</table>

Indexing algorithm takes between 5~12 minutes to index large projects with at least 75,000 files and 20,000 revisions. Also, because runtime computation is limited to index lookups, indexing algorithm shows fast response time, e.g. less than three seconds for types with more than 600 revisions (RQ1).

### 3.4.2 Controlled User Study

The goal of our controlled user study is to determine whether the temporal dimension that Tempura adds to Eclipse can help developers learn about code history more quickly and accurately (RQ2). While a more long-term study is better suited to accurately evaluate Tempura since it’s main purpose is to extend an IDE with code history information, we conducted a small scale study as a preliminary demonstration of Tempura’s usability and efficacy.
We conducted a between-group user study with 10 participants. They were randomly divided into two groups, a control and a treatment. We gave the participants a code base that they were not familiar with, and asked them to answer questions about the history of the code base. The participants in the control group used only EGit, and those in the treatment group used only Tempura to explore code history. EGit follows the conventional approach of separating VCS operations and programming, in much the same way as other VCS plugins (e.g., Subversive). We conducted a between-group study rather than an in-group study because once participants learn the history of the subject program using either EGit or Tempura, they cannot re-learn it and produce fair results.

A. Study Design

We used a Java project called LANSimulation from the Refactoring Lab Session exercise developed at LORE [DRVRDB] and used in several previous user studies [LCJ13, DMJN07, DMJN08, DNMJ08]. While the original LANSimulation project is small with only 5 classes, we believe that it is of a reasonable scale for subjects to understand and work with in a short period of time. The study involved two sessions, with the entire study lasting about 1 hour. During the first session, participants were given the a version of LANSimulation project, adopted and modified from the original LANSimulation project, and asked to study and understand the code base in 15 minutes. In the second session immediately following the first, participants were given the same project that has undergone 20 revisions. We built the LANSimulation project's history by making a set of systematic changes (Table 3.3), mainly refactorings adopted from the LORE exercise, interspersed with non-code changes (e.g., formatting). Participants answered a set of questions regarding the changes (Table 3.4, given in a text file). Both groups were given a written user guide for the tools they used prior to the the user study [Lee, Ro], and were also allowed to refer to the user guides at any point during the user study. The author of this dissertation was present during each participant’s user study, but did not interact with the participants. There were no time restriction for the second session, and participants were allowed to answer the questions in any order.

To answer RQ2, we scored participants’ answers following a clearly defined rubric
Table 3.3: Changes made to LANSimulation project.

1. Encapsulate fields in Message class
2. Encapsulate fields in Node class
3. Non-code changes
4. Extract a new method called log in Network class
5. Rename Message class to Packet
6. Non-code changes
7. Inline printAccounting method in Network class
8. Add getter and setter methods for firstNode field, and getter method for workstations field in Network class
9. Move DefaultExample method from Network class to LANSimulation class
10. Move log method from Network class to Node class
11. Move printDocument method from Network class to Node class
12. Non-code changes
13. Add Printer and Workstation classes that extend Node class, and remove type field from Node class
14. Extract isAtDestination method in Network class
15. Rename printDocument method in Node class to printJobStatus
16. Add LANSimulationUtil.jar that contains NetworkPrinter hierarchy, and deprecate previous print methods in Network class
17. Fix assertEquals calls in LANTests class
18. Clean up try-catch statements in LANTests class
19. Non-code changes
20. Add empty test methods for testing simple, XML, and HTML print functions

Table 3.4: Questions given to user study subjects.

1. What happened to the Message class?
2. What happened to the private Network.printAccounting method?
3. What happened to the Network.printDocument method?
4. Can you identify any other methods that were previously defined in Network class?
5. What are the changes made to/in the Node class?
6. Implement the bodies of testPrint, testHTMLPrint, and testXMLPrint methods in the LANTests class.

(Table 3.5) and measured the time it took for participants to answer the question. We scored each participant’s answers without knowing to which group the participant belonged. Each user study session was recorded using a screencast software, and the recordings were analyzed after the study to determine the time that participants spent using the designated tools. The usage of the tools were marked by any window or interface of the tools being in focus. We concentrated on the tool usage time as opposed to the time it took for participants to finish the user study in order to eliminate as much variables as possible, for example, participants’ experiences with Eclipse and speed of programming. We also calculated the rate of information acquirement by dividing the
Table 3.5: Grading rubric

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Renamed to Packet (2pts), Other changes (1pt)</td>
</tr>
<tr>
<td>2.</td>
<td>Inlined (2pts), Other changes (1pt)</td>
</tr>
<tr>
<td>3.</td>
<td>Moved from Network to Node (1pt) Renamed (1pt)</td>
</tr>
<tr>
<td>4.</td>
<td>DefaultExample (1pt), log (1pts)</td>
</tr>
<tr>
<td>5.</td>
<td>Encapsulation (1pt)</td>
</tr>
<tr>
<td>6.</td>
<td>log from Network (1pt) Added NetworkPrinter hierarchy (1pt) printDocument from Network (1pt) printJobStatus (renamed from printDocument) (1pt)</td>
</tr>
<tr>
<td>6.</td>
<td>Implement test methods using NetworkPrinter classes (2pt per test method)</td>
</tr>
</tbody>
</table>

Maximum possible score: 21

raw score by tool usage time, to obtain a more precise indication of how efficiently the tools help developers gain understanding of code history.

One of the participants was a professional software engineer, and the rest of the participants were graduate students in the computer science department at the University of Illinois at Urbana-Champaign, majoring in various sub-disciplines. Participants had at least three years of Java experience, with seven participants having more than eight years of experience. Five participants indicated that they use Eclipse IDE for their programming tasks, one uses IntelliJ, and the rest do not use IDEs regularly. All participants had at least two years of experiences using VCSs (Git, SVN, or Mercurial). The control and treatment groups had similar average years of programming experience (7.2 years and 7.4 years, respectively), but the control group had overall more experience with VCS (5.8 years) then the treatment group (3.8 years). Also, three out of five participants in the treatment group stated that they do not use IDEs regularly, whereas the control group had one non-IDE user.

Participation was strictly voluntary with no rewards offered, and invitations to the study were sent through individual emails and departmental mailing lists.

B. Results

Table 3.6 shows the user study results from 10 participants. The Score (%) columns show the scores each participant received for their answers, and the Time (s) columns show
Table 3.6: User study results.

<table>
<thead>
<tr>
<th></th>
<th>Control - EGit</th>
<th>Treatment - Tempura</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARTIC Score (%)</td>
<td>28.6</td>
<td>42.9</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1387</td>
<td>595</td>
</tr>
<tr>
<td>Score per min.</td>
<td>0.26</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C2</td>
<td>57.1</td>
<td>66.7</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1341</td>
<td>1202</td>
</tr>
<tr>
<td>Score per min.</td>
<td>0.54</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3</td>
<td>42.9</td>
<td>66.7</td>
</tr>
<tr>
<td>Time (s)</td>
<td>961</td>
<td>1393</td>
</tr>
<tr>
<td>Score per min.</td>
<td>0.56</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C4</td>
<td>52.5</td>
<td>76.2</td>
</tr>
<tr>
<td>Time (s)</td>
<td>797</td>
<td>1384</td>
</tr>
<tr>
<td>Score per min.</td>
<td>0.83</td>
<td>0.69</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C5</td>
<td>57.1</td>
<td>71.4</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1521</td>
<td>843</td>
</tr>
<tr>
<td>Score per min.</td>
<td>0.47</td>
<td>1.07</td>
</tr>
<tr>
<td>AVG.</td>
<td>47.6</td>
<td>64.8</td>
</tr>
<tr>
<td>Time (s)</td>
<td>1201</td>
<td>1083</td>
</tr>
<tr>
<td>Score per min.</td>
<td>0.53</td>
<td>0.79</td>
</tr>
</tbody>
</table>

the tool usage time in seconds. The Score per min. columns show the rate of information acquirement, calculated in terms of raw score that each participant gained per minute. Participants also used Tempura for shorter period of time than EGit, suggesting that they were able to learn about code history more quickly with Tempura. The higher average rate of information acquirement for participants using Tempura also corroborates this conjecture.

On average, the participants using Tempura scored 17pp higher with 50% higher efficiency than the participants using EGit (RQ2).

We performed a one-way t-Test to calculate the p-values for the score, tool usage time, and the rate of information acquirement, which are shown in the table below. We chose the one-way test because Tempura extends Eclipse’s existing programming support features with code history information that are attainable through conventional VCS plugins. We predicted that if a developer is experienced in Eclipse and VCS plugins, using Tempura will only increase the developer’s efficiency and accuracy in learning code history. Our prediction for RQ2 was that the treatment group will have a higher mean score, and that the control group will have a longer mean study duration.

<table>
<thead>
<tr>
<th>Score (%)</th>
<th>Time (s)</th>
<th>Score per min.</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = 0.03097</td>
<td>0.2941</td>
<td>0.0351</td>
</tr>
</tbody>
</table>

While the sample size is not large enough to show statistical significance in general, the p-values for the score and rate of information acquirement show some statistical
significance of the collected data (<0.05). More importantly, the p-values provide a promising outlook for our approach of extending IDEs with a temporal dimension, by demonstrating efficiency and utility of Tempura.

Overall, participants using Tempura were able to get higher scores, answer questions faster, and especially have a higher rate of score per time unit.

C. Post Study Survey

We asked all user study participants to fill out surveys about the tools they used.

For the participants in the control group, four out of five stated that they use Eclipse IDE, and of those four, three participants stated that they mainly use Git VCS. However, none of them uses EGit, and instead prefer to use command-line interface for Git. One of those participants (C4) said:

[EGit’s] Interface is hard to use and does not offer much over CLI [command-line interface].

While Tempura does not aim to support all the Git operations one can use in a command-line interface, participants found Tempura’s Open Type in History feature to be useful, which closely resemble Git’s log and show operations that can be targeted to a specified file (rather than type). Tempura’s Open Type in History feature provides developers with only the commits that changed a specific type, and also displays the contents of the file that contains that type in every revision. Participants T1 and T2 from the treatment group respectively said:

It was easy to group changes for a class together and easily find edits, addition, removals.

I found myself using type lookup in history a lot, mostly because I could isolate the changes made to a specific class in history.

Participants in the treatment group also noted that Tempura would be useful in a team environment. Participants T2 and T5 respectively said:
I suspect this tool would be more useful for programmers in industry who are assigned onto an existing project with a lot of history.

[...] it would be very useful in a multiperson programming environment.

All data collected from the user study, including the answers, and surveys from each participant are publicly available at http://mir.cs.illinois.edu/tempura.

3.5 Threats to Validity

The main threat to internal validity is the completeness of the temporal dimension that Tempura implements. While the decision to add the temporal dimension to the code completion and type search features was made based on authors’ personal experiences about the usefulness of these features and discussions with colleagues with extensive Eclipse experiences, it is likely that other IDE features can also benefit from addition of the temporal dimension. Also, user study participants were given a choice to either use their own machines for familiarity or use a designated machine for convenience. These machines varied greatly in terms of specifications and operating systems, which could have affected the computation time of Eclipse, EGit, and Tempura. Lastly, because the participants in the treatment group were aware that Tempura is a new addition to Eclipse, it may have biased them toward or against it.

The main threats to external validity are the degree to which the experiments and user study scenarios are representative of the target population and practice. First, while the Eclipse projects used in our experiment for evaluating scalability are widely used by many developers, it may not be representative of all repositories used for Java projects. Second, we analyzed the results from only 10 study participants, and all but one were graduate students in the computer science department at the University of Illinois at Urbana-Champaign. Although collectively they have diverse experiences in Java, Eclipse, and Git, they may not be representative of the target population. Third, the LANSimulation project used in the user study is of small scale in terms of size and complexity, and the changes made to the project were mainly refactorings interspersed with superflu-
ous formatting or comment changes. While these were deliberately and carefully made choices in order to ensure that participants can complete their studies roughly within one hour, the size and complexity of the project as well as the nature of the changes may not represent the daily programming tasks of professional developers. Lastly, Tempura is an Eclipse plugin for Java projects with Git VCS, and was evaluated only against Eclipse's EGit plugin. While many other IDEs follow the convention of handling only the latest version of code and leaving code history to separate VCS tools, which was the main problem that Tempura is designed to resolve, differences between IDEs and their VCS tools could require different implementations of the temporal dimension extension and thus lead to different study results.
Chapter 4

Related Work

This chapter presents an overview of various work by other researchers that are related to the contributions of this dissertation.

4.1 Drag-and-Drop

Drag-and-drop interfaces have traditionally been used in visual programming environments such as Alice [Con97], EToys [onln] and Scratch [MRR+10]. In such environments, novice programmers write programs using visual blocks instead of text. Programmers use drag-and-drop as the primary means for organizing and restructuring those visual blocks.

Because visual blocks can be clunky to navigate in large programs, we eschew this approach in DNDRefactoring and implemented it directly in the textual Java editor, Package Explorer, and Outline View. Moreover, simple restructuring of visual blocks merely moves blocks to different locations in the program without considering behavior preservation. DNDRefactoring, on the other hand, intuitively maps each drag-and-drop operation to a corresponding refactoring operation that, when performed, preserves program behavior.

The typical modal window-based approach to invoking and configuring refactorings was introduced in the first refactoring tool, i.e. the Refactoring Browser [RBJ97]. For more than a decade, little has changed in the interface of refactoring tools. Recently, Murphy-Hill et al. introduced new approaches to invocation with selection assists [MHB08a] and gesture-to-refacto-ring mappings [MHAB11]. Eclipse and IntelliJ have also introduced in-place refactoring features [onlg] that allow widely-used refactorings to be configured directly in the editor without the need for a modal window.
Commercial tools such as CodeRush with Refactor! Pro also aid programmers’ refactoring tasks with suggestions and visual hints within the code, without modal windows. Nonetheless, these new approaches still rely exclusively on keyboard shortcuts and mouse menus. Our work investigates and demonstrates the potential of new methods of invocation for refactoring tools.

While drag-and-drop infrastructure has always been available in modern IDEs, none have truly exploited its capabilities. Existing IDEs such as Eclipse, NetBeans and IntelliJ provide minimal support for drag-and-drop refactoring. Currently, the only refactoring supported is Move refactoring, which can be invoked by drag-and-dropping a class into a package in the Outline View. All other drag-and-drop operations are interpreted as plain textual moves. Existing products dedicated to restructuring code only target organizational refactorings between different packages. For instance, Restructurer101 provides a graphical view of all the classes and packages in the system and allows a developer to perform Move refactorings on them via drag-and-drop. To the best of our knowledge, our tool is the first to leverage the drag-and-drop as an intuitive way to invoke a variety of refactorings beyond Move refactorings.

4.2 Code Completion and Navigation

There are many research efforts concentrating on extending IDE features such as code completion and navigation or applying information stored in VCSs to software engineering processes.

Much work has focused on improving code completion. For example, Omar et al. developed a system architecture that allows library developers to introduce interactive interfaces, called palettes, for library users to use for code completion in the context of class instantiation. However, palettes are highly susceptible to changes. If the code for which palettes are implemented is modified, the palettes will also need to be modified. Perelman et al. defined a language of partial expressions that makes type-directed predictions to help developers find method names based on the given arguments, arguments based on the method name, or to complete binary expressions such
as assignment statements [PGBG12]. Similarly, Duala-Ekoko and Robillard [DER11] developed a tool called API Explorer that helps developers discover API methods or types that are inaccessible from a given API type, by leveraging the structural relationships between API elements. Such tools help developers use the unknown APIs, but they do not help developers with the APIs they used to know but have changed, and as such, the existing tools would be useful for stable APIs, but the development process in general is inherently dynamic where the code and APIs change constantly. Tempura focuses on the change.

Other researchers have focused on providing predictive support for code completion. Muşlu et al. [MBH+12] introduced an Eclipse plugin called Quick Fix Scout, that computes on behalf of developers the consequences of Quick Fix recommendations. Quick Fix Scout allows developers to remove compilation errors faster, but it does little to help developers rebuild their mental model of the code that may have become outdated and caused the compilation error. Learning from the history of code through temporal code completion, on the other hand, can help developers build a concrete and complete mental model of the program which may even help reduce mistakes that Quick Fix Scout aims to reduce. Predictive support can also be interpreted in terms of providing the code completion proposals that developers are most likely to select. Mooty et al. introduced Calcite [MFSM10] which extends the existing code completion in Eclipse with crowdsourcing to support completion of object instantiation (i.e. constructor completion). Calcite uses a database containing the most common ways to construct objects, built by mining example code on the web, and uses the web search hit frequency to rank the completion proposals. Calcite helps developers learn from the crowdsourced information, but it is possible that such information do not pertain to every developer's code. For example, the most commonly used instantiation method found on the web may not conform to their coding standard or style. In contrast, the source of temporal code completion's proposals is the history of the code itself.

Code navigation is also an active research topic. Ko et al. [KMCA06] reported that developers engaged in software maintenance tasks spent up to 35% of their time navigating through the code, learning how the code works and how to modify it to complete
their tasks. It is not difficult to conjecture that the time spent in navigating the code will only increase if developers have to switch between versions. Other researchers have also examined ways to minimize context switch. For example, Janzen and De Volder introduced JQuery [JDV03], an Eclipse plug-in browser tool based on logic query language. JQuery allows users to form specialized browsers in which to navigate code and to perform queries, providing an explicit and unbroken representation of the exploration paths. This helps reducing the cognitive burden of retaining navigation context for users. Similarly, Storey et al. combined the notion of waypoints and social tagging in their Eclipse-plugin called TagSEA [SCS+07]. TagSEA allows developers to add Javadoc style tags in their code that are shared with other developers, which they can use to search, group, manage and filter related code. Their approach allows developers to implicitly create a simple navigational structure. We believe Tempura achieves similar benefits by removing the time delimitation when programming.

4.3 Software Evolution

LaToza et al. [LVD06] reported that 50% of developers find understanding the history of a piece of code to be a difficult problem. As such, many researchers have extended code completion and navigation tools with varying interpretations of historical information.

Robbes and Lanza [RL08a, RL10] used change-based information to improve code completion, comparing all the code completion proposals that were suggested and the one that was selected at every step in the development history. They collected historical information such as the last modified or added date of a class/method, and used it to rank proposals in their tool. The modifications they consider, however, only pertain to the body of methods or classes, so deleted elements, moved, or renamed methods or classes are disregarded. Bruch et al. [BMM09] introduced an intelligent code completion system that calculates each proposal’s relevance in a given context, by using examples found in existing code repositories, and uses the information to filter and rank the proposals. While the system helps developers to focus only on relevant API elements, it disregards deleted elements that can no longer be relevant in the current version, effectively hiding
parts of code evolution information. Such tools utilizing historical information can be useful, but they effectively compress the entire history for the current (single) version of code. Tempura, in contrast, allows developers to explore any version in the history.

Similarly, for navigation, Singer et al. [SES05] introduced NavTracks, a tool that monitors and analyzes the navigation history of software developers as they perform their tasks, forming associations between related files. These associations are used to recommend potentially related files when, for example, a developer opens a file that she knows is relevant to a bug fix. Mäder and Egyed [ME11] implemented and evaluated a program editor tool with code navigation feature augmented with requirements traceability, which allows developers to quickly identify where a requirement is implemented. While improving the speed and accuracy of development tasks, these tools still only work on one version of the code at a time. Tempura, on the other hand, allows developers to navigate to older versions of (read-only) code just as they can navigate the current version. We believe such history navigation will greatly improve the efficiency of developers’ tasks by allowing them to quickly access any code from the past.

Other researchers have considered different and novel ways of promoting integration of VCSs into IDEs. Researchers found that merge conflicts are frequent and persistent, and introduced tools that continuously perform speculative merges in order to detect conflicts as soon as possible [BHEN11a, BHEN11b, GaS12]. Such approaches and tools achieve a tighter integration of IDEs and VCSs, but they focus on merge conflict resolution which is strictly a VCS operation. In contrast, Tempura aims for even tighter integration where code evolution information stored in VCS becomes a part of the IDEs and thus a part of everyday development process.

There also have been research efforts focusing on IDEs instead of VCSs as the source of code evolution information [RL07, RL08b, NVC+12, HS10]. Researchers developed change monitoring and tracking tool for IDEs that capture code changes and programming operations at a finer granularity. These research projects focus on change-level software evolution, where changes are treated as the first-class object. Tempura, in contrast, treats history as the first-class object. Our goal is to provide developers with code evolution information that they can use immediately, and the commit-level information
can provide more succinct information. Change-level information is much more detailed, but it can also be excessive and overwhelming for developers. For example, a developer looking for a deleted method may not care about how many times a certain refactoring was invoked and canceled to remove the method. Such detailed information can be useful in identifying characteristics and patterns of changes, but it needs to be studied and analyzed before it can be useful for developers.

Some empirical research provides a good motivation for our approach of extending code completion and navigation with the temporal dimension. Code completion support in IDEs only shows public (or otherwise accessible) identifiers in other classes. Dig and Johnson found that 80% of changes that break client applications are caused by API-level refactorings [DJ05]. Kim et al. also found that there is an increase in number of bug fixes after API level refactorings, often caused by mistakes in applying refactorings and behavior modifying edits together [KCK11].
Chapter 5

Conclusions and Future Work

While IDEs provide a broad set of facilities that aid many aspects of software development process. Most modern IDEs support the plug-in concept which allows developers to further extend and customize the IDEs. However, we believe that the plug-in concept is encouraging IDEs to become simply a collection of disparate tools, each with its own complex interfaces and operations. This creates a burden on developers who not only have to write their code but also has to learn to use the tools. The contributions of this dissertation tackles these problems with a new class of tools that are more intuitive and instinctive by leveraging developers’ inherent understanding of their code and software development activities. We achieve this by extending an IDE with new yet familiar dimensions. Drag-and-Drop Refactoring adds a tactile dimension to the refactoring tools’ interfaces, enabling developers to perform refactorings by moving program elements directly, thus bypassing complex dialog-based invocation and configurations. Tempura adds a temporal dimension to code completion and navigation functions in IDEs, allowing developers to search for deleted program elements through code completion proposals and type search instead of manually switching between versions and searching through code history in VCSs.

5.1 Future Work

Both Drag-and-Drop Refactoring and Tempura have a number of possible improvements and extensions.
5.1.1 Drag-and-Drop Refactoring

The current implementation of DNDRefactoring assumes that programmers can accurately distinguish between different program elements. We believe selection assist tools such as [MHB08a] will be an effective complement to DNDRefactoring. Also, visual cues such as highlights or tooltips indicating the specific refactoring that will be invoked may help narrow down programmers’ selection of drop targets.

Apart from possible functional features for future versions of DNDRefactoring, we plan to apply the idea of programming by gestures and actions in different aspects of software engineering and evaluate its effectiveness. Since the action of drag-and-drop is more intimate and interactive, we conjecture that the use of DNDRefactoring during pair programming will be very helpful. During pair programming, an agile software development technique where two programmers work together at one workstation, the driver obviously has more control over the code changes than the observer. While this is expected, the driver's action of drag-and-drop may be easier and more intuitive for the observer to follow and understand. We also believe the drag-and-drop refactoring would be an effective tool in teaching refactoring. We are interested in impact of the difference in perception of the program – as a malleable entity instead of textual representation of a program – when novice programmers learn refactoring.

One of the critiques we received from user study participants and colleagues was the fact that some programmers are less inclined to use a mouse during programming. We see this as no strict limitation of our tool, but we recognize it as a possible barrier for some programmers to adopt DNDRefactoring. One possible remedy to this issue is to utilize a completely new technology, one of which is a touch screen. Eclipse, as of the Indigo version, does not support touch screen functionality, but we see a potential in implementing DNDRefactoring for the touch screen interface. We hypothesize that touch screens will provide even more intimate and hands-on programming experience.

Lastly, we plan to conduct a long-term study to analyze and evaluate the utility of DNDRefactoring in assisting programmers with floss refactorings. Would programmers using DNDRefactoring use the refactoring tool in IDE more often? If so, what kind of
refactorings would they use DNDRefactoring for? We plan to collect refactoring data from programmers using DNDRefactoring in the wild, using such tools as [VCN+12], and study the impact of DNDRefactoring on floss refactoring.

5.1.2 Tempura

We asked the user study participants for additional features they would like to use, and suggestions included simplified comparisons between revisions and refactoring detections:

“[...] it would be nice to have diffing at the API level, essentially just showing the two eclipse ‘outlines' for the two different versions.”

“It would be great if there was a way to quickly tell from where a method was moved.”

These suggestions corroborate supporting refactoring inference that Tempura currently only partially provide, and we plan to extend the refactoring inference support in the future.

As discussed previously, the Tempura's support for the temporal dimension may also be extended. For example, we have implemented proof-of-concept feature called Open Call Hierarchy in History that extends Eclipse's Open Call Hierarchy, which finds all the previous callers of the selected member even if they no longer call it, or even if the selected member is now deleted and no longer present in the code base. Such a feature may be useful, for example, when an incorrectly-implemented method was inlined into its callers. Developers would greatly benefit from being able to quickly find and fix the callers that now have the method inlined. While the feature is not mature yet, we plan to improve and support even more temporal features that can benefit developers in their everyday development tasks. Similarly, the temporal dimension could also be extended into the future. For example, Tempura currently explores the history of code, but predicting possible merging of features would have an interesting impact on improving collaboration.
While we have identified and handled some cases where changes in history impact temporal code completion results (Section 3.3.3), there could potentially be an unlimited ways the changes can affect the results. For example, a receiver type may have had a more complex history than simply extending and unextending a past superclass. It would therefore be beneficial if such changes that impact temporal code completion results can be automatically categorized and identified.

Other commonly used Git commands could have significant impact on Tempura's operations. For example, Tempura currently does not handle in any special way when a commit is reverted in Git, and while the indices can be recomputed when a commit is reverted, a more efficient way of handling Git's revert, or any other changes that impact the commits, would be beneficial for Tempura.

Lastly, a more extensive and thorough evaluation of Tempura is needed. While our experiment with three large Eclipse projects showed scalability of Tempura, a more rigorous evaluation of Tempura's performance and scalability is desirable. In addition, our controlled user study demonstrated that Tempura can help developers learn the code history efficiently and quickly, but there are other potential benefits of Tempura that we wish to evaluate more thoroughly. For example, we believe Tempura will help prevent developers from having to switch context between programming and searching in the history, which can drastically improve their efficiency. Our controlled user study, however, required minimal programming from participants, because we wanted to focus on the learning of code history. We plan to conduct a long-term study with developers out in the wild, in order to evaluate Tempura's impact on developers' daily programming tasks.

In addition to possible extensions for Drag-and-Drop Refactoring and Tempura, the ideas presented in this dissertation could also be extended by exploring novel dimensions for IDEs. For example, allowing developers to see what their teammates are working on real-time through a new visual dimension of read-only editors may have an interesting impact on their productivity.
References


