Tempura: Temporal Dimension for IDEs

Yun Young Lee, Darko Marinov, Ralph E. Johnson

Department of Computer Science, University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {lee467,marinov,rjohnson}@illinois.edu

***Abstract*—Modern integrated development environments (IDEs) make many software engineering tasks easier by providing automated programming support such as code completion and navigation. However, such support – and therefore IDEs as a whole – operate on *one revision of the code at a time*, and leave handling of code history to external tools or plugins, such as EGit for Eclipse. For example, when a method is removed from a class, developers can no longer find the method through code completion. This forces developers to manually switch across different revisions or resort to using external tools when they need to learn about previous code revisions.**

We propose a novel approach of adding a temporal dimension to IDEs, enabling code completion and navigation to operate on *multiple revisions of code at a time*. We previously introduced the idea of temporal code completion and navigation, and presented a vision for how that idea may be realized. This paper realizes that vision by implementing and evaluating a prototype tool called Tempura. We describe our algorithm for processing and indexing historical code information from repositories for Tempura, and demonstrate Tempura's scalability with three large Eclipse projects. We also evaluate Tempura's usability through a controlled user study. The study participants learned about the code history with more accuracy when using Tempura compared to EGit. Although the sample size was not large enough to provide strong statistical significance, the results show a promising outlook for our approach.

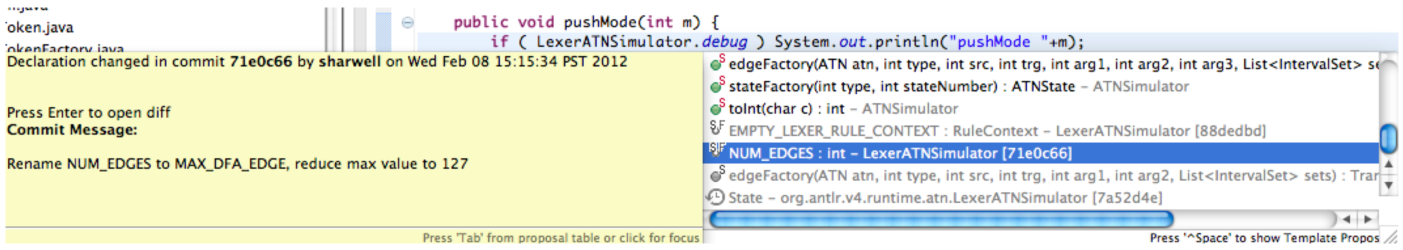
I. Introduction

Modern integrated development environments (IDEs) provide automated programming support that make many software development tasks easier. For example, many IDEs offer context-specific programming assistance 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 [1] 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 [2]. Some research prototypes additionally use dynamic program information to improve navigation [3], [4].

While such automated programming support in IDEs help developers' programming tasks, these features, and thus the IDEs as a whole, operate on *one revision* of the code at a time. However, developers working on continuously evolving projects not only have to work with the most current revision 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 [5]. 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, 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 pertinent 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 core IDE features like code completion and navigation and VCS features. Current IDEs that restrict core IDE features to operate only on one revision of code inherently hamper developers' productivity, because developers are forced to tediously search through revision history in VCSs or manually switch to different revisions when seeking information from past revisions of code.

For example, 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 revision 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` itself may have undergone many changes, or been deleted altogether. Also, if the commit messages are unclear, or if the commits contain multiple unrelated changes, Alice's task becomes even more complicated and tedious. Her search process will likely become slower as the size, duration, and number of people involved in her project increases. 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. However, if Alice can still find the `NUM_EDGES` field in the



```
    public void pushMode(int m) {
        if ( LexerATNSimulator.debug ) System.out.println("pushMode "+m);
```

Declaration changed in commit **71e0c66** 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

```
edgeFactory(ATN atn, int type, int src, int trg, int arg1, int arg2, int arg3, List<IntervalSet> se
stateFactory(int type, int stateNumber) : ATNState – ATNSimulator
toInt(char c) : int – ATNSimulator
EMPTY_LEXER_RULE_CONTEXT : RuleContext – LexerATNSimulator [88dedbd]
NUM_EDGES : int – LexerATNSimulator [71e0c66]
edgeFactory(ATN atn, int type, int src, int trg, int arg1, int arg2, List<IntervalSet> sets) : Trar
State – org.antlr.v4.runtime.atn.LexerATNSimulator [7a52d4e]
```

oken.java
okenFactory.java
Press 'Tab' from proposal table or click for focus
Press '^Space' to show Template Propos

Fig. 1. Tempura's historical code completion proposals for `LexerATNSimulator` class are shown in gray, with historical information displayed in tooltip. Example code is from `ANTLR4` project.

completion proposal list and use it to pinpoint the exact change that removed the field, she could complete her task much more efficiently.

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 revisions* at a time without resorting to manual revision switching. This paper builds on our short paper that introduced the idea of temporal code completion and navigation [6]. Our approach locates types and members from past revisions of code that are relevant to the current context and presents them through code completion. 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]. This paper makes the following contributions:

- An algorithm for processing and indexing past revisions of code in VCSs in order to support temporal dimension in IDEs.
- An open-source prototype Eclipse plugin, Tempura, that embodies our approach for temporal dimension, supporting the Java programming language and Git VCS.
- An evaluation of Tempura that answers 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 evaluate how Tempura compares with EGit that handles code history separately from the current revision of code (RQ2). The results show that Tempura can index the history of projects with over 80,000 files and 20,000 revisions in less than 12 minutes, with less than 3 second run-time response time, and that Tempura allows developers to learn about code changes with 36% higher accuracy and with 50% higher efficiency in terms of rate of information acquirement compared to EGit.

[1]Tempura is a Japanese dish, but we derived the name from the word "temporal" for our tool.

All materials, including Tempura's source code and user study materials, are publicly available at http://mir.cs.illinois.edu/tempura, and more details of our work can be found in [7].

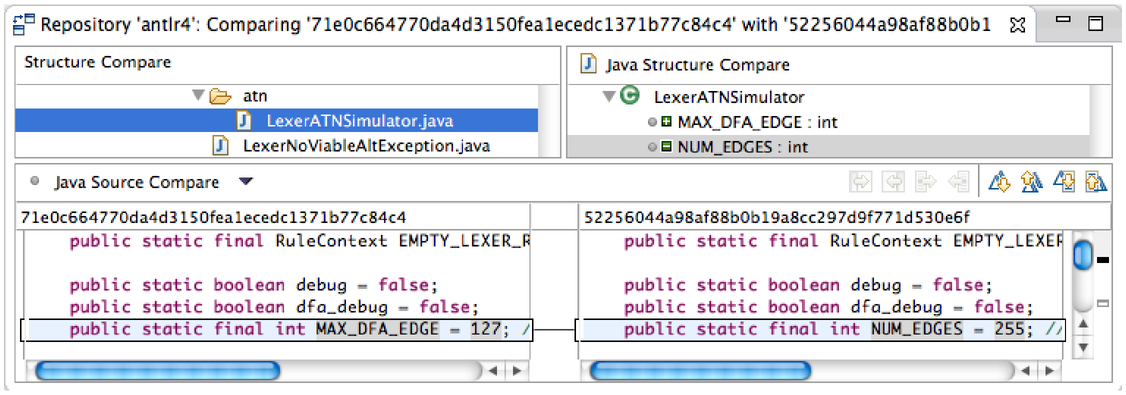
II. Tempura Tool

Tempura embodies our approach of extending IDEs with a temporal dimension by allowing Eclipse to simultaneously operate on previous revisions of the code as well as the current revision. Tempura processes and indexes historical API 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. Tempura supports one Git repository at a time, requiring an Eclipse workspace to have projects from a single Git repository.

A. Temporal Code Completion

Tempura augments Eclipse's code completion with proposals that were possible in all the previous revisions of the code in the given context. The context is defined by program elements involved in the code completion invocation, their package and inheritance relationships, and the access restriction rules between Java program elements. More precisely, let us call the element on which code completion is invoked the *receiver element* (a static type reference of `LexerATNSimulator` type in our example), and the element from which the code completion was invoked the *caller element* (the `pushMode` method in Figure). The *receiver element* may be a reference variable that can be resolved, or a string token that can be used as a prefix of a name. The *caller element* can be a method or a type. The resolved type or name of the *receiver element* and the resolved type of the *caller element*'s (enclosing) type, together with their package and inheritance relationships, form the code completion context C.

Figure shows the code completion proposals for the `LexerATNSImulator` class, where historical proposals are displayed in gray. Each historical proposal item also displays relevant information from the VCS in its tooltip, including the date, author, message, and ID of the commit (Git uses SHA-1 hash for commit ID) that removed the particular method or field. 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. Therefore, when a developer selects a historical proposal, Tempura displays a comparison (or a *diff view*) between the revision that last contained the historical proposal


Repository 'antlr4': Comparing '71e0c664770da4d3150fea1ecedc1371b77c84c4' with '52256044a98af88b0b1

Structure Compare

atn

LexerATNSimulator.java

LexerNoViableAltException.java

Java Structure Compare

LexerATNSimulator

MAX_DFA_EDGE : int

NUM_EDGES : int

Java Source Compare

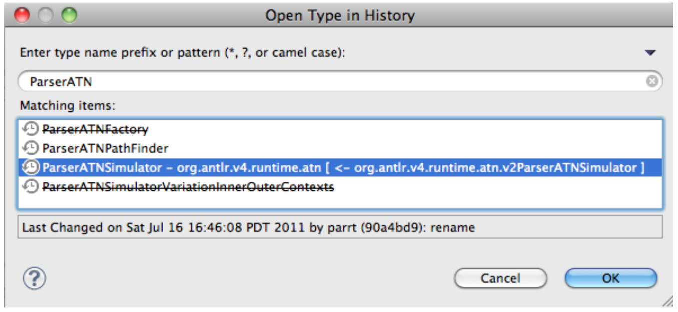
```
71e0c664770da4d3150fea1ecedc1371b77c84c4
    public static final RuleContext EMPTY_LEXER_R

    public static boolean debug = false;
    public static boolean dfa_debug = false;
    public static final int MAX_DFA_EDGE = 127; /
```

```
52256044a98af88b0b19a8cc297d9f771d530e6f
    public static final RuleContext EMPTY_LEXER

    public static boolean debug = false;
    public static boolean dfa_debug = false;
    public static final int NUM_EDGES = 255; //
```

Fig. 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).


Open Type in History

Enter type name prefix or pattern (*, ?, or camel case):

ParserATN

Matching items:

~~ParserATNFactory~~

ParserATNPathFinder

ParserATNSimulator – org.antlr.v4.runtime.atn [<– org.antlr.v4.runtime.atn.v2ParserATNSimulator]

~~ParserATNSimulatorVariationInnerOuterContexts~~

Last Changed on Sat Jul 16 16:46:08 PDT 2011 by parrt (90a4bd9): rename

Fig. 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. Tempura also identifies and displays the renamed or moved types with arrows.

and the revision that removed it (Figure I). 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.

B. Temporal Code Navigation

Tempura supports temporal code navigation by allowing developers to search for and open any type from any revision of their projects using the *Open Type in History* dialog (Figure II-B), including deleted types that are no longer present in the current revision. 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 II-B 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 revision. Selecting a type from the search result displays at the bottom of the dialog window the date and the revision ID of the commit that last changed the selected type. In addition, Tempura identified that ParserATNSimulator was renamed from a type called v2ParserATNSimulator, and describes the change with an arrow depicting the rename (or move) change (Section III-A).

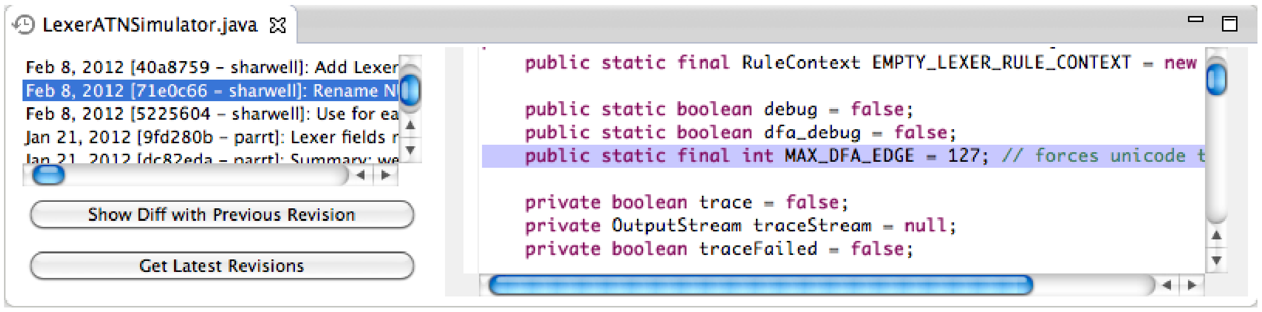
If the developer chooses to open a type from the dialog, Tempura opens the type in a *read-only historical editor* (Figure 4). The historical editor contains a list of revisions in which the file containing the historical type was modified, along with date, commit ID, author, and commit message. The editor also uses background colors to show changes with respect to a previous revision. For example, Figure 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 I). Similarly, green background highlights added lines of code. Tempura also allows developers to open a diff view comparing the selected revision in the list and its parent revision (Figure I) with the "Show Diff with Previous Revision" button.

III. Algorithm

The goal of Tempura is to present developers with all the code completion proposals and type search results that were possible at some point in time in the past. The most straight-forward way of collecting the historical proposals for a context C is to check out every revision of a project and invoke code completion in C in each revision, and similarly for collecting type search results. However, this approach proved to be prohibitively expensive as our initial experiment with the ANTLR4 project [8] took roughly 30 seconds to check out and build each revision of the project. There were total of 1636 revisions of the project at the time of experiment, which would have translated to over 13 hours to process all revisions. In addition, Eclipse allows developers to use different build systems other than its own, and it is impossible for Tempura to support every possible build system. Our goal for Tempura is to make it usable in terms of computation efficiency and portability, therefore we implemented an algorithm that can operate independently of any build system.

Our algorithm is a two-step algorithm that (1) parses every revision of every class from the VCS and indexes API information of types and members by (declaring) types' fully-qualified names (FQNs), and (2) retrieves and filters code completion proposals and type search results.

1) Parsing Past Revisions of Code: Different VCSs track changes in different ways. VCSs such as CVS and Subversion store information as a list of file-based changes, and track changes made to each file over time. In contrast, Git stores a snapshot of added or modified files every time a commit is made. For files that have not changed, Git stores a link to the



```
Feb 8, 2012 [40a8759 – sharwell]: Add Lexer
Feb 8, 2012 [71e0c66 – sharwell]: Rename N
Feb 8, 2012 [5225604 – sharwell]: Use for ea
Jan 21, 2012 [9fd280b – parrt]: Lexer fields r
Jan 21, 2012 [dc82eda – parrt]: Summary: we
```

```
public static final RuleContext EMPTY_LEXER_RULE_CONTEXT = new

public static boolean debug = false;
public static boolean dfa_debug = false;
public static final int MAX_DFA_EDGE = 127; // forces unicode t

private boolean trace = false;
private OutputStream traceStream = null;
private boolean traceFailed = false;
```

Fig. 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).

previous identical file it has already stored for efficiency [9]. Note that this difference does not preclude other VCSs or make Git more advantageous for Tempura. Our decision to support Git was based on its popularity, and we believe supporting other VCSs would require only minor implementation changes. The following pseudo code describes how we parse historical code revisions.

```
function processRepository(Repository)
  foreach commit C in a Repository
      in chronological order
      foreach file A that is added in this commit
          parse(A)
      foreach file M that is modified in this commit
          parse(post-modification revision of M)
      foreach file D that is deleted in this commit
          parse(pre-deletion revision of D)
      foreach file R that is renamed in this commit
          //Git-specific
          parse(post-rename revision of R)

function parse(File)
  //parse compilation unit in File with ASTVisitor
  foreach top-level Class, Interface, or
   Enum declaration T in File
     store commit data {commit ID, File's next commit ID,
     path of File, and File ID}, T's FQN,
     and FQNs of T's superclass and interfaces
  foreach nested Class, Interface, or
   Enum declaration T in File
     store commit data, T's FQN, T's enclosing type's FQN,
     and FQNs of T's superclass and interfaces
  foreach Method declaration M in File
      store commit data, access modifier,
      return type's FQN, method signature,
      and M's enclosing type's FQN
  foreach Field declaration F in File
      store commit data, access modifier, type' FQN,
      and F's enclosing type's FQN
```

Tempura uses Eclipse's Java parser to parse the added, modified, deleted, or renamed files in each commit from a project's Git repository. The rename changes are detected by using Git's rename detection capability, and Section III-A details how Tempura uses it to identify renamed or moved types. Tempura extracts API information from each file, i.e., declarations of types defined in a file, and their method, field, and inner class declarations (and their members recursively). The parsed API information is stored in a data object called `HistoryElement`, which is a simplified abstract syntax tree (AST) node object that also records the ID of the commit (SHA of the commit object) in which the file containing the type or member was modified, ID of the file's next commit (child revision in the current branch), ID of the file (SHA of the blob object), and the path of the file (this is to efficiently support VCSs which track files, not individual classes). Tempura will store the `HistoryElements` in a set indexed by the (enclosing) type's FQN.

Parsing a Java file every time it is added, modified, renamed, or deleted effectively records for each type or member 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 child revision, `71e0c66`, then renamed the field to `MAX_DFA_EDGE`, effectively removing the `NUM_EDGES` field from the class (Figure I). Therefore, by also recording the child revision's ID, Tempura can easily and quickly show a diff view when a historical proposal is selected. In addition, the parsing and indexing of the API information allows Tempura to use the information for both code completion and type search.

2) Retrieving and Filtering Proposals: For code completion, when a developer invokes code completion in a context C, Tempura uses Eclipse JDT's code completion to resolve and identify the caller and receiver types. Tempura then uses the receiver type's FQN to find the set of `HistoryElements`. While traversing the set of `HistoryElements`, Tempura uses the caller and receiver types' identities to compute access rules and filter the `HistoryElements`. For example, any `HistoryElement` with private access modifier is excluded unless the caller and receiver types are the same type. In addition, Tempura disregards proposals that exist in the current revision of the receiver type so as not to duplicate the proposals. For type search, Tempura matches the search phrase to the FQNs of all the types indexed from the repository. Unlike temporal code completion, however, temporal navigation with type search is not restricted to any context, and also includes the types that are present in the current revision of code along with deleted types. This is because the read-only historical editor allows developers to choose any revision of the type using the list of revisions on the left-hand side. Limiting runtime computations to simple index lookup and filtering for code completion and type search allows for a fast response time regardless of the length of project or receiver type's history.

A. Challenges

There are a number of important aspects in presenting code history through the current workspace with code completion and navigation that require careful consideration. We describe what they are and how Tempura handles them.

Handling Complex Changes: 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 their resolved types that determine accessible program elements. Those program elements, however, could also have had complicated history. 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 cases and their solutions.

Firstly, consider a type T that extends a super type T' in the current revision. When a developer invokes code completion on an element E of type 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 be ignoring some parts of the code history. Tempura handles the possible changes in inheritance relationships by recording a type's super type and interfaces during indexing. Then, while 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. Any changes to T'', or more specifically deletion of its members, *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.

Secondly, changes in non-identifying components of an element can also affect the result of temporal code completion. For example, a type in Java is identified by its FQN, which does not include the value of the type's 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 simply with their signatures during indexing presents some limitations. For example, while access modifiers and return type are not part of a method's signature, any change in them affects the resulting set of code completion proposals. Tempura therefore records such components when 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, two instances of F will be indexed with T's FQN, one with `protected` access modifier and the other with `public` access modifier.

Lastly, there may be cases where seemingly identical elements in different revisions may in fact be different. For example, consider a type T that had a method with the following signature `setLocalTime(LocalTime t)` in revision `r1`. In a later revision `r2`, the type T was modified where an import statement `import java.time.LocalTime;` was changed to `org.joda.time.LocalTime;`. While the change in import statements clearly changes the signature of the `setLocalTime` method, simple parsing cannot identify the change of the argument `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 uses the simple name of the types. If the simple name is not found in the import statements or the `java.lang` package, and there is no wildcard import statement, then Tempura uses the declaring type's package name to resolve types.

Supporting Branches: Code history of large projects is rarely linear. They involve multiple branches throughout their life cycle, and they present various challenges when merging the branches. 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 [10]. Multiple branches and their merging also pose an important issue when presenting code history information to developers, as presenting a code completion proposal or type search result that pertains only to a different branch can confuse developers and lead them to build wrong implicit knowledge of their code. Tempura therefore maintains a separate 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 index only if it is. A separate index for each branch is necessary because an index effectively compresses a branch's history, and the timestamp on each commit conveys no information about which branch it belongs to.

Handling merge commits also present challenges for Tempura during indexing, most importantly because a non-conflicting 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, a field added to a class 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 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, skipping merge commits may result in loss of 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 conflicting or non-conflicting would be trivial, identifying for a past merge commit may require extra computation (e.g. re-merging of the involved branches) to determine the conflict status.

Inferring Changes: While inferring the nature of changes, specifically refactorings [10-14], 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 byte content similarity in percentage required to pair a deleted and an added files in a commit as a renamed (or moved) file. Tempura sets the threshold to a conservative 99. When parsing, Tempura keeps a record of pairs of paths indicating pre- and post-rename (or move) files, and uses the path pairs to identify the FQNs of pre- and post-rename (or move) classes (including non-public and inner classes declared in a file). This information is then displayed to developers in the *Open Type in History* dialog (Figure II-B). Because the rename/move detection is based on Git's byte comparison and not on syntactic and semantic analysis of Java code, 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

TABLE I. INDEXING OF EVERY JAVA FILE IN EACH REVISION FROM GIT REPOSITORIES.

Project	# of Commits	Time (s)	# of Files Parsed	Parsed Data Size (bytes)	File Size (bytes)
org.eclipse.jdt.ui	26684	308.732	118225	1486824832	41287550
org.eclipse.platform.ui	25052	237.29	102567	1259510656	42480222
org.eclipse.jdt.core	21165	711.92	83194	4661719552	24077726
ANTLR4	1636	28.20	7781	108760168	4979603
LANSimulation	21	0.96	54	416672	15813

TABLE II. TEMPORAL CODE COMPLETION INVOCATIONS.

Class	# Revisions	# Hist. Proposals	Time (s)
org.eclipse.jdt.core.JavaCore	712	599	2.08
org.eclipse.jdt.internal.compiler.problem.ProblemReporter	611	1385	1.30
org.eclipse.jdt.internal.compiler.parser.Parser	556	1699	1.36
org.eclipse.jdt.internal.compiler.lookup.ReferenceBinding	212	1229	2.58
org.eclipse.jdt.internal.compiler.Compiler	125	58	0.57
org.antlr.v4.runtime.atn.LexerATNSimulator	107	200	0.58

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' simple name before and after the change.

Tempura limits refactoring inference only to class rename and moves, because inferring refactorings on members 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 [11], and proposed a heuristic-based algorithm that detects renamed methods between two revisions of code [12]. Tempura may be extended to leverage existing research tools to infer refactorings in the future. For example, Negara et al.'s method of assigning unique IDs to every AST node when tracking changes [13] suggests a promising approach.

IV. 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 [14], a widely used Git plugin for Eclipse, in helping developers learn about code history. Through both evaluations, we answer the following 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 code completion and navigation features that are common in IDEs help developers to learn code history more accurately and efficiently?

A. Indexing and Runtime Efficiency

To answer **RQ1**, we evaluated Tempura's efficiency in indexing 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 I. 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 (used in our user study, described in Section IV-B) for references. The indexing takes place when Tempura is first installed, and subsequent revisions are parsed immediately following a commit to the repository, thus incurring only negligible cost. Our experiments showed that Tempura can index the code history of even large-scale projects with more than 20,000 revisions in less than 12 minutes, which we believe demonstrates Tempura's efficiency and scalability.

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 II). 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 in this paper. 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. With further caching, subsequent code completion invocations showed sub-second response times, and they can be further reduced, for example, by limiting the number of past revisions to inspect.

One-time indexing takes between 5~12 minutes for large projects with over 80,000 files and 20,000 revisions. Also, because runtime computation is limited to index lookups, our algorithm shows fast response time, e.g., under three seconds for types with more than 600 revisions (RQ1).

B. 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 its 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 project that they

TABLE III. 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 `Netowork` 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 IV. QUESTIONS GIVEN TO USER STUDY SUBJECTS.

1. What happened to `Message` class?
2. What happened to private `Network.printAccounting` method?
3. What happened to `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 `Node` class?
6. Implement the bodies of `testPrint`, `testHTMLPrint`, and `testXMLPrint` methods in `LANTests` class.

were not familiar with, and asked them to answer questions about its code history. 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 unlearn it to produce fair results.

1) Study Design: We used a Java project called `LANSimulation` from the Refactoring Lab Session exercise developed at LORE [15] and used in several previous user studies [16], [17], [18], [19]. 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 a revision of `LANSimulation` project, adopted and modified from the original `LANSimulation` project, and asked to study and understand the code in 15 minutes. In the second session immediately following the first, participants were given the same project that has undergone 20 revisions. The first paper author built the `LANSimulation` project's history by making a set of systematic changes (Table III), 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 IV, given in a text file). Both groups were given a written user guide for the tools they used prior to the the user study [20], [21], and were also allowed to refer to the user guides at any point during the user study. There were no time restrictions for the second session, and participants were allowed to answer the questions in any order.

TABLE V. GRADING RUBRIC.

1. Renamed to `Packet` (2pts), Other changes (1pt)
2. Inlined (2pts), Other changes (1pt)
3. Moved from `Network` to `Node` (1pt)
Renamed (1pt)
4. `DefaultExample` (1pt), `log` (1pts)
5. Encapsulation (1pt)
`log` from `Network` (1pt)
Added `NetworkPrinter` hierarchy (1pt)
`printDocument` from `Network` (1pt)
`printJobStatus` (renamed from `printDocument`) (1pt)
6. Implement test methods using `NetworkPrinter` classes (2pt per test method)
Maximum possible score: 21

To answer **RQ2**, we scored participants' answers following a clearly defined rubric (Table V) and measured the time it took for participants to answer the questions using the designated tools. We then 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 uses 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 that each participant took 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 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. All 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) than 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.

TABLE VI. User study results.

Control - EGit				Treatment - Tempura			
Participant	Score (%)	Time (s)	Score per min.	Participant	Score (%)	Time (s)	Score per min.
C1	28.6	1387	0.26	T1	42.9	595	0.91
C2	57.1	1341	0.54	T2	66.7	1202	0.70
C3	42.9	961	0.56	T3	66.7	1393	0.60
C4	52.5	797	0.83	T4	76.2	1384	0.69
C5	57.1	1521	0.47	T5	71.4	843	1.07
AVG.	47.6	1201	0.53	AVG.	64.8	1083	0.79

2) Results: Table VI 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 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. Our results show a higher average rate of information acquirement for participants using Tempura than those using EGit, suggesting Tempura enabled them to learn about code history more quickly.

On average, the participants using Tempura scored 36% higher accuracy with 50% higher efficiency than the participants using EGit (RQ2).

We performed several statistical analyses on the scores and tool usage times to verify their statistical significance. The p-values for the score and time from each test are shown in the table below.

Statistical Analysis	Score P-values	Time P-values
Welch's t-test	0.03097	0.2941
Kolmogorov-Smirnov	0.04076	0.8187
Exact Bootstrap	0.03250	0.5759

Since we hypothesized that participants using Tempura would earn higher scores than those using EGit, we tested the null hypothesis versus the one-sided alternative hypothesis for the scores data. The first test we performed is Welchs two-sample t-test. As shown in the table, this test resulted in a statistically significant P-value of roughly 0.03 (<0.05). Thus, we conclude that the one-sided alternative hypothesis is true, that the mean score using Tempura is greater than the mean score using EGit. However, while Welchs two-sample t-test is robust and is especially designed for small sample sizes, it does require that the populations from which the data are collected be normally distributed, and normality is difficult to verify with small sample sizes.

Therefore, we also tested these hypotheses using two other analyses, to serve as checks. The first of these analyses is the two-sample Kolmogorov-Smirnov test, which is a non-parametric test that does not place any distribution assumption on the underlying population. The p-values from the Kolmogorov-Smirnov test show similar results as the Welch's t-test, with the p-value of scores of roughly 0.04. Thus we conclude that the mean score for those using Tempura is greater than the mean score for those using EGit. The second of these analyses is the Exact Bootstrap test, which allows for the simulation of thousands of new data sets that, when taken as a whole, can be used to estimate the unknown underlying population. More precisely, the Exact Bootstrap test examines every possible permutation of each of our data set ($5^5 = 3125$), and compares the mean of each of the control bootstrap datasets with that of the treatment bootstrap dataset, resulting in $3125 \times 3125 = 9765625$ comparisons. Again, the Exact Bootstrap test resulted in the p-value for the score being roughly 0.03, confirming the results of and conclusions from the first two tests. Since the mean score was found to be greater for participants using Tempura, but no difference was found between the mean times to completion, it stands to reason that the mean rate of information acquirement (score per minute) is also greater for participants using Tempura. More importantly, however, these p-values provide a promising outlook for our approach of extending IDEs with a temporal dimension, by demonstrating efficiency and utility of Tempura.

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.

V. 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 we only had 10 participants in our study, we cannot trust that random assignment of participants to the treatment and control groups would be sufficient to make the groups homogeneous. The small sample size also prevented us from blocking on possible confounding variables such as machine used, job, Java experience, IDE experience, and VCS experience.

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. Firstly, while the Eclipse projects used in our experiment for evaluating scalability are widely used by many developers, they may not be representative of all repositories used for Java projects. Secondly, 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. Thirdly, 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 superfluous formatting or comment changes. While these were deliberately and carefully made choices 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 revision 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.

VI. Future Work

Firstly, we asked the user study participants for additional features they would like to have in Tempura, 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 provides; extending the refactoring inference support is potential future work.

Secondly, 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. 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.

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.

VII. Related Work

A. Code Completion and Navigation

Many researchers have 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 [22]. 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 [23]. Similarly, Duala-Ekoko and Robillard [24] 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. While such tools help developers use the *unknown* APIs, 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.

Other researchers have focused on providing predictive support for code completion. Muşlu et al. [25] 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. 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 [26] which extends the existing code completion in Eclipse with crowdsourcing to support completion of object instantiation (i.e., constructor completion). While Calcite helps developers learn from the crowdsourced information, such information may 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. [27] 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 revisions. Other researchers have examined ways to minimize context switch between programming and navigation. For example, Janzen and De Volder introduced JQuery [28], 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. Similarly, Storey et al. combined the notion of waypoints and social tagging in their Eclipse-plugin called TagSEA [29]. 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. DeLine et al. introduced an overview features to a development environment called Code Thumbnails that allows developers to form and use a spatial memory of source code, both within a file and between multiple files [30]. We believe Tempura achieves similar benefits by removing the time delimitation when programming.

B. Software Evolution

LaToza et al. [5] 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 [31], [32] 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. [33] 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 revision and therefore hides parts of code evolution information.

Similarly, for navigation, Singer et al. [34] 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 [35] 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 revision* of the code at a time.

Other researchers have considered different and novel ways of promoting integration of VCSs into IDEs [36], [37], [38], [39]. For example, 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. While such approaches and tools achieve a tighter integration of IDEs and VCSs, 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 [13], [40], [41], [42]. 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.

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 [43]. 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 [44].

VIII. Conclusions

We introduced a novel approach of adding a *temporal dimension* to IDEs, by seamlessly extending IDEs' common programming support such as code completion and navigation with information from code history. Tempura, our prototype Eclipse plugin, realizes this vision by extracting API information from a Java project's Git repository and presenting the information through code completion and navigation, even for classes, methods, or fields that no longer exist in the current revision. We evaluated Tempura in two ways. First, in order to determine scalability of Tempura with large projects, we used Tempura to compute historical information from three Eclipse projects, each with more than 75,000 files and about 20,000 revisions. Tempura completed one-time-only indexing for the projects in 5~12 minutes. In addition, Tempura showed fast response time, for example, less than three seconds even for types with more than 600 revisions. Second, we conducted a between-group controlled user study that compared Tempura and EGit. Participants not only learned about code history more quickly and efficiently with Tempura, but also 36% more accurately, although some of these improvements were not statistically significant due to small sample size.

Acknowledgment

We would like to thank Sam Harwell, Johnston Jiaa, and Sarfraz Khurshid for their contribution to the initial idea and tool. We thank Yun Hee Lee, Jeff Overbey, Mohsen Vakilian, and Roshanak Zilouchian for piloting our user study, and Nikhil Unni and Andrey Zaytsev for reviewing our drafts. We also thank Nathan Hirtz for his help and advices on statistical analyses, and all the user study participants for their time. The authors also thank the anonymous reviewers for their insightful reviews. This work is partially supported by the National Science Foundation under grant numbers CCF-1012759, CCF-1117960, and CCF-1439957.

References

[1] "Eclipse Code Recommenders," http://eclipse.org/recommenders/.

[2] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete Completion Using Types and Weights," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2013.

[3] D. Rothlisberger, M. Harry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz, "Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks," *IEEE Trans. Softw. Eng.*, 2012.

[4] M. Härry, "Augmenting Eclipse with Dynamic Information," Master's thesis, University of Bern, 2010.

[5] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining Mental Models: A Study of Developer Work Habits," in *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[6] Y. Y. Lee, S. Harwell, S. Khurshid, and D. Marinov, "Temporal Code Completion and Navigation," in *Proceedings of the 2013 International Conference on Software Engineering, New Ideas and Emerging Results*.

[7] Y. Y. Lee, "Towards Multidimensional Integrated Development Environment for Improved Productivity," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 2014.

[8] "ANTLR," www.antlr.org.

[9] S. Chacon, *Pro Git*, 1st ed. Berkeley, CA, USA: Apress, 2009.

[10] A. Tarvo, T. Zimmermann, and J. Czerwonka, "An Integration Resolution Algorithm for Mining Multiple Branches in Version Control Systems," in *Proceedings of the 27th IEEE International Conference on Software Maintenance*, 2011.

[11] P. Weissgerber and S. Diehl, "Identifying Refactorings from Source-Code Changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006.

[12] S. Kim, K. Pan, and E. J. Whitehead, Jr., "When Functions Change Their Names: Automatic Detection of Origin Relationships," in *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005.

[13] S. Negara, M. Vakilian, N. Chen, R. E. Johnson, and D. Dig, "Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?" in *Proceedings of the 26th European Conference on Object-Oriented Programming*, 2012.

[14] "EGit," https://www.eclipse.org/egit/.

[15] S. Demeyer, M. Rieger, B. Van Rompaey, and B. Du Bois, "Refactoring Lab Session," http://lore.ua.ac.be/Research/Artefacts/refactoringLabSession/.

[16] Y. Y. Lee, N. Chen, and R. E. Johnson, "Drag-and-drop Refactoring: Intuitive and Efficient Program Transformation," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[17] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen, "Refactoring-Aware Configuration Management for Object-Oriented Programs," in *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[18] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective Software Merging in the Presence of Object-Oriented Refactorings," *IEEE Trans. Softw. Eng.*, 2008.

[19] D. Dig, S. Negara, V. Mohindra, and R. Johnson, "ReBA: Refactoring-aware Binary Adaptation of Evolving Libraries," in *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[20] Y. Y. Lee, "Tempura User Guide," http://mir.cs.illinois.edu/yunyounglee/TempuraUserGuide.pdf.

[21] R. Rosenberg, "EGit User Guide," https://wiki.eclipse.org/EGit/User_Guide#Git_Perspective_and_Views.

[22] C. Omar, Y. Yoon, T. D. LaToza, and B. A. Myers, "Active Code Completion," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[23] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, "Type-directed Completion of Partial Expressions," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.

[24] E. Duala-Ekoko and M. P. Robillard, "Using Structure-based Recommendations to Facilitate Discoverability in APIs," in *Proceedings of the 25th European Conference on Object-oriented Programming*, 2011.

[25] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative Analysis of Integrated Development Environment Recommendations," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, 2012.

[26] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing Code Completion for Constructors Using Crowds," in *Proceedings of the 2010 IEEE Symposium on Visual Languages and Human-Centric Computing*, 2010.

[27] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks," *IEEE Trans. Softw. Eng.*, 2006.

[28] D. Janzen and K. De Volder, "Navigating and Querying Code Without Getting Lost," in *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*, 2003.

[29] M.-A. D. Storey, L.-T. Cheng, J. Singer, M. J. Muller, D. Myers, and J. Ryall, "How Programmers Can Turn Comments into Waypoints for Code Navigation," in *Proceedings of the 24th IEEE International Conference on Software Maintenance*, 2007.

[30] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. Drucker, and G. Robertson, "Code Thumbnails: Using Spatial Memory to Navigate Source Code," in *Proceedings of the Visual Languages and Human-Centric Computing*, 2006.

[31] R. Robbes and M. Lanza, "How Program History Can Improve Code Completion," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[32] ——, "Improving Code Completion with Program History," *Automated Software Engg.*, vol. 17, no. 2, 2010.

[33] M. Bruch, M. Monperrus, and M. Mezini, "Learning from Examples to Improve Code Completion Systems," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2009.

[34] J. Singer, R. Elves, and M.-A. Storey, "NavTracks: Supporting Navigation in Software Maintenance," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.

[35] P. Mäder and A. Egyed, "Do Software Engineers Benefit from Source Code Navigation with Traceability? - An experiment in Software Change Management," in *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering*, 2011.

[36] Y. Yoon, B. A. Myers, and S. Koo, "Visualization of fine-grained code change history." in *Proceedings of the Visual Languages and Human-Centric Computing*, 2013.

[37] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Crystal: Precise and Unobtrusive Conflict Warnings," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[38] ——, "Proactive Detection of Collaboration Conflicts," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, 2011.

[39] M. L. Guimarães and A. R. Silva, "Improving Early Detection of Software Merge Conflicts," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[40] R. Robbes and M. Lanza, "A Change-based Approach to Software Evolution," *Electron. Notes Theor. Comput. Sci.*, vol. 166, 2007.

[41] ——, "SpyWare: A Change-Aware Development Toolset," in *Proceedings of the 30th international conference on Software engineering*, 2008.

[42] S. Hayashi and M. Saeki, "Recording Finer-grained Software Evolution with IDE: An Annotation-based Approach," in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 2010.

[43] D. Dig and R. Johnson, "The Role of Refactorings in API Evolution," in *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.

[44] M. Kim, D. Cai, and S. Kim, "An Empirical Investigation into the Role of API-level Refactorings During Software Evolution," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.