Capturing and Exploiting IDE Interactions

Zhongxian Gu[†] Drew Schleck[†] Earl T. Barr[£] Zhendong Su[†]

[†]Department of Computer Science
University of California, Davis
{zgu, dtschleck, su}@ucdavis.edu

[£]Department of Computer Science
University College London
e.barr@ucl.ac.uk

Abstract

Integrated development environments (IDEs) dominate the production and maintenance of software. Developers interact intensively with their IDEs while working. These interactions reflect a developer's thought process and work habits. By capturing and exploiting comprehensive, fine-grained IDE interactions, we can build intelligent IDEs that improve programmer productivity. This next generation of IDEs will incorporate a general framework to capture and exploit IDE interactions, and create an ecosystem of developer-aware applications and plugins. IDE++ realizes this framework on top of the popular Eclipse IDE and can be downloaded from the Eclipse marketplace. To demonstrate IDE++'s comprehensive and granular capture of interactions, we capture, then faithfully replay, a developer's IDE actions on six nontrivial programming tasks. We built four applications upon IDE++ to illustrate 1) the need for capturing comprehensive, fine-grained IDE interactions, and 2) the promise of developer-aware IDEs.

Categories and Subject Descriptors: D.2.3 [SWE]: Coding Tools and Techniques

General Terms: Human Factors, Languages, Measurement

Keywords: User-monitoring, integrated development environment

1. Introduction

Development and maintenance dominate the cost of software [2]. 97% of .NET developers use Microsoft Visual Studio and 73% of Java developers use Eclipse-based IDEs [12]. Thus, increasing the utility and power of IDEs will improve programmer productivity and reduce the cost of software. Interactions between a developer and her IDE capture how the developer writes a piece of code and reflect her thought process and work habits. When we monitor a programmer's IDE interactions, we can look for patterns in the interaction stream that indicate she needs help and provide instant assistance. If we detect that a developer is unsure about which API to use, we can recommend an API and show relevant examples. An IDE can also adapt itself to a programmer's work habits: if it detects that a developer habitually runs test cases after an editing session, it could run relevant tests automatically.

We believe the key to building the next generation of IDEs is to transform IDEs from being order-takers into intelligent, user-aware programs that monitor and reason about how their users interact with them, like the original conception and underlying technology, not the unfortunate realization, of Office Assistant in Microsoft Office [13]. We envision establishing an ecosystem of IDE applications and extensions that exploit this awareness to dynamically personalize their interface, to teach their developers how to use them more effectively, to help them follow best practices, and to point out features that are likely to be relevant to a developer's current task.

Developer-aware IDE Applications An ecosystem of behavior-aware IDE applications will benefit the users of IDEs, those who study developers and software processes, and IDE developers. Programmers are often confronted with unexpected, repetitive tasks, like conflict resolution during version control or protocol updates after an API change. A behavior-aware application could identify these cases and suggest macros. Developer-aware navigation could infer landmarks from the developer's, or her colleague's, behavior, such as frequently returning to a particular class or method, then speed subsequent navigation by jumping to those landmarks [6]. Advocates of the quantified-self movement claim that one can improve oneself through self-study [31]. Developer-aware IDEs will allow developers to discover that conditions under which they, personally, are

Onward! 2014, October 20–24, 2014, Portland, OR, USA.
Copyright © 2014 ACM 978-1-4503-3210-1/14/10. . . $15.00.
http://dx.doi.org/10.1145/2661136.2661144

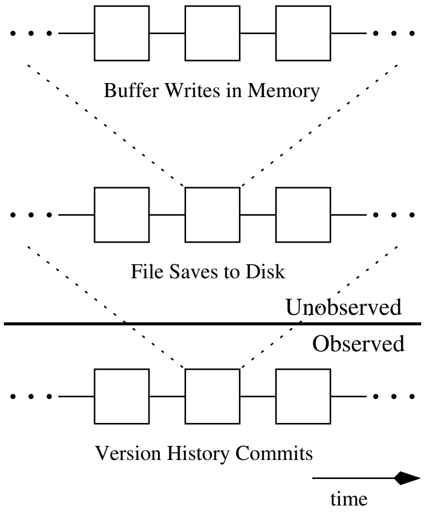


Figure 1: IDE++ captures what is currently unobserved.

likely to introduce bugs, perhaps right after they return from a coffee break. An experienced programmer's interaction log can teach a novice how best to work with an API, refactor a method, fix a bug, or debug a race condition. When editing an unfamiliar file, a programmer may wish to know which files other developers who edited that file had also opened and which methods had spent the most time on screen. Finally, IDE developers themselves can examine interaction logs to learn which features users actually use to more rapidly improve their IDE's UI.

Capturing currently unobserved IDE interactions will also provide empirical software engineering a rich, new source of data to mine. Interaction logs, suitably sanitized to avoid a Hawthorne effect [28], will allow researchers to investigate questions such as "How do the interaction histories of experienced programmers differ from those of novices?" and "Are there correlations between IDE interactions and bug introduction or cost overruns?". For instance, "Are developers more likely to introduce bugs when interrupted?" where IDE++ could allow a researcher to define an interrupt to be a large, relative to the median, pause between edits. Consider Figure 1. At the base, it shows commits to a version control history, whose mining, due to the increasing importance of open source and its adoption of version control, has transformed empirical software engineering since 2000. Above we see a developer's local file writes; and above that the developer's buffers writes. Both of these are currently unobserved; IDE++ allows their capture. In short, IDE++ and a cohort of cooperative developers willing to share their interactions will give researchers new purchase on existing problems [11] and open the door to pose and answer new questions.

Capturing IDE Interactions While our work is the first to advocate the systematic monitoring of all kinds of IDE interactions, prior efforts exist that focus on monitoring a subset of interactions [22, 26, 32, 36]. Some behavior-aware applications also already exist. The Mylyn Monitor, proposed by Kersten and Murphy, intercepts programmer UI and command interactions to track Eclipse's layout and dynamically reconfigure it to support task-focused workflows [16]. The Mylyn Monitor focuses on task-related interactions, such as launches of and switches between views, (*i.e.* GUI windows in Eclipse). Like other developer-aware applications, the Mylyn Monitor does not consistently, comprehensively capture fine-grained events, like edits at the granularity of keystrokes or selections in different contexts. In contrast, we advocate and realize the comprehensive and fine-grained capture of IDE interactions to make it easy to build unanticipated developer-aware applications.

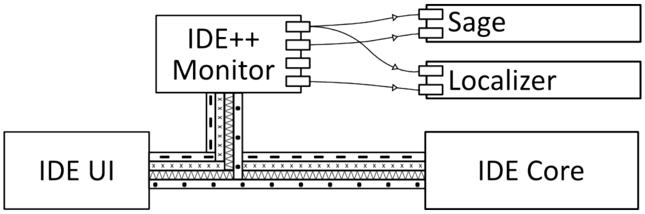


Figure 2: To support developer-aware applications, IDE++ captures and republishes interactions between a developer and an IDE.

Of course, a developer would describe her interaction with an IDE at a fairly high level of abstraction relative to the underlying sequence of raw, low-level, hardware events. For instance, developers might describe the IDE interactions of a coding session as consisting of a sequence of editing, browsing, testing, and debugging tasks. The appropriate level of abstraction will vary with the development task and technology changes. It is for this reason that we advocate fine-grained interaction capture: we do wish to preclude any conceivable developer-aware IDE application. This work focuses on interaction capture, setting the stage for, but largely leaving to future work, the task of interpreting and exploiting these interactions. Of course, IDE interactions cannot support arbitrary inferences. When a huge gap occurs between two user commands, we cannot infer why that gap occurred, nor why a developer chose to accomplish a task in a particular order. In short, we do not capture coffee runs.

We have developed IDE++ on top of Eclipse. It consists of an infrastructure that monitors fine-grained programmer interactions in Eclipse and tool support to write custom, developer-aware applications. IDE++'s architecture is publish and subscribe, as Figure 2 shows. IDE++ monitors and extracts IDE interactions, then publishes them to registered applications.

We developed four applications — DevTime, Sage, Proctor and Localizer — to illustrate the promise of an ecosystem of developer-aware application built on IDE++ and the necessity of capturing comprehensive, fine-grained interactions.

DevTime summarizes all of a developer's IDE interactions and faciliates introspective improvement of one's use of an IDE; its utility rests on comprehensive event capture.

Sage teaches novices how to use Eclipse's built-in features to be more productive; the opportunity to use some of these features can only be detected by fine-grained event capture, such as of the keystrokes that comprise manual commenting.

Proctor helps to identify which methods programmers should consider testing after an editing session; it collates events from the test UI and editor and demonstrates the need for comprehensive capture.

Localizer uses fine-grained code edits to help programmers determine where their unit test cases fail.

This paper makes the following main contributions:

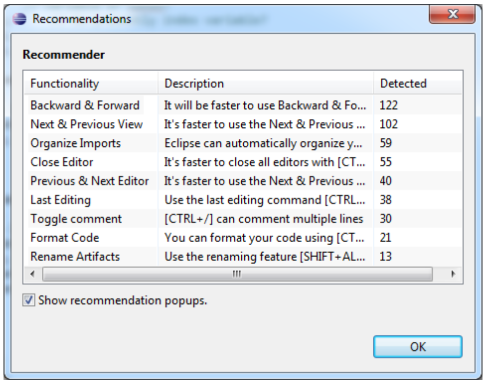
- We advocate the systematic capture and utilization of IDE interactions as the basis of a new class of developer-aware IDE applications;
- We present the IDE++ monitoring infrastructure, built for Eclipse, that comprehensively captures and republishes fine-grained programmer interactions; and
- To demonstrate the promise of the ecosystem of developer-aware IDE applications that IDE++ makes possible, we built four applications that require and exploit IDE++'s comprehensive, fine-grained IDE interactions.

The early feedback on IDE++ has been positive and is evidence for its potential for supporting useful customized applications. We have released IDE++ to the Eclipse marketplace. Tutorials and downloads are available at `http://marketplace.eclipse.org/content/ide#.UzGgj1FdV3c`.

The rest of the paper is structured as follows. Section 2 presents Sage, an application whose intuitive utility illustrates the promise of the developer-aware applications that IDE++ makes possible. Section 3 specifies which interactions we capture and the methodology used to identify them. Next, Section 4 describes the design and implementation of IDE++. In general, it is difficult for one person to flawlessly mimic another. To demonstrate that IDE++'s interaction capture is comprehensive and fine-grained, Section 4 shows that one programmer is able to fully replay the IDE session of another programmer, then illustrates the ease of writing applications in IDE++. Finally, Section 5 discuss related work and Section 6 concludes.

2. Illustrative Example

An IDE is a complicated program, with a lengthy learning curve [17]. Novice IDE users are often unaware of shortcuts [27]. Even programmers who have mastered an IDE may use their IDE inefficiently. For example, a programmer who does not know that Eclipse's Organize-Import feature (hotkey: Shift+Ctrl+O) automatically inserts import statements, must manually type import statements. Although these features are well documented, searching for and reading that documentation often distracts from a developer's current task.



Functionality	Description	Detected
Backward & Forward	It will be faster to use Backward & Fo...	122
Next & Previous View	It's faster to use the Next & Previous ...	102
Organize Imports	Eclipse can automatically organize y...	59
Close Editor	It's faster to close all editors with [CT...	55
Previous & Next Editor	It's faster to use the Next & Previous ...	40
Last Editing	Use the last editing command [CTRL...	38
Toggle comment	[CTRL+/] can comment multiple lines	30
Format Code	You can format your code using [CT...	21
Rename Artifacts	Use the renaming feature [SHIFT+AL...	13

☑ Show recommendation popups.

Figure 3: Sage records how often a programmer could have used a built-in feature, but did not.

An efficient learning strategy is to have an expert who looks over your shoulder and tells you how to accomplish a task more efficiently. Sage acts as just such an expert. It monitors a programmer's IDE interactions and pops up a tip showing the functionality that would have been faster. For example, Sage pops up a tip teaching the user about the Toggle-Comment feature (hotkey: Ctrl+/), if it detects that a programmer is manually commenting several lines of code. Sage focuses on hotkey usage; Spyglass also monitors an IDE user, but focuses on code navigation tasks [33]. Sage also records the number of times a user could have used a feature but did not. A programmer who uses mouse clicks to switch editors could examine its output in Figure 3 and see that using Eclipse's Backward and Forward commands might be more efficient for him.

Sage is an example of applying a finite-state machine (FSM) to the interaction stream for self-improvement. To build it, we identified Eclipse features that beginners neglect, manually discovered how to accomplish the task each of these feature speeds up and replaces, then encoded that feature-bypassing sequence of actions into an FSM. While Eclipse is running, Sage continuously feeds the programmer's interactions into each supported feature's automaton. When an automaton reaches a final state, Sage displays the related tip notification. Currently, Sage supports 11 features, shown in Figure 3. These features are spread across different domains such as editing, refactoring, and browsing; their use saves keystrokes and can improve productivity

3. Design and Implementation of IDE++

We integrated IDE++ into Eclipse because it is the dominant IDE for Java programmers. We first describe the methodology we used to realize our goal of comprehensive and fine-grained interaction capture in IDE++. We then describe IDE++'s

Component	Views
General	Bookmarks, Console, Error Log, Markers, Navigator, Outline, Problems, Progress, Project Explorer, Properties, Search, Tasks, Templates
Development	Editor, Call Hierarchy, Declaration, Hierarchy, Javadoc, Package Explorer
Browsing	Members, Packages, Projects, Types
Debug	Breakpoints, Debug, Display, Expressions, Memory, Modules, Registers, Variables
Testing	Unit testing

Table 1: The components and their views of the default configuration of Eclipse.

architecture, how its clients can subscribe to its events and extend it to capture events from new plugins.

3.1 Methodology

Programmers interact with IDEs through GUI windows called views. Every interaction is tied to a view. Components group views with the same purpose. As an example, Table 1 lists all the components, and their constituent views, in Eclipse's default configuration. Different views define disjoint sets of interactions. Programmers can edit code in the editor view, not a search view. A programmer interacts with an IDE in one view at one time. He can switch to another view by performing a special interaction view switch such as opening the JUnit view or issuing the Previous-View command. Sometimes view switching is implicit: running a test case after editing a program switches the view from editing to testing.

To ensure that IDE++ captures comprehensive and fine-grained interactions, we first sought to identify all core Eclipse JDT views, then, for each view, all interactions defined by that view. In both cases, we systematically studied the Eclipse GUI, its documentation, and, finally, its source code. In particular, when seeking to identify all the interactions in a view, we reasoned from first principles, asking ourselves what interactions a particular view *must* have in order to achieve the goal for which it was designed. We began our search for Eclipse interaction with the GUI components in Eclipse's default configuration.

Take the editor view as an example. The most fine-grained interactions we capture are file buffer changes: inserting or removing one character in a file. Some fine-grained interactions might construct a high level interaction. For example, code changes also reflect language-specific semantics, such as AST changes. For example, a sequence of insertions might reduce to adding a field to a class. Our model captures these high-level changes.

Eclipse's JUnit framework provides support for running tests inside of Eclipse. After systematically analyzing it as described above, we determined it would be best to extend their `TestRunListener` class. This notifies us of the user starting and ending a JUnit test session, along with notifications each time a test case starts and finishes. Components supported in Eclipse is a moving target. Currently, we support all the views in default configuration for Java development.

Of course, we cannot know the set of IDE interactions precisely, both because of differences among IDEs and because, as technology advances, IDEs will acquire new features that define new actions and views. Indeed, we restricted IDE++ to Eclipse's Java functionality to affirm the concept of comprehensive, fine-grained IDE event capture. Keeping pace with Eclipse for Java, let alone extending it to other Eclipse personalities and to other IDEs, will require substantial engineering effort. IDE++ is open source; if enough people find it useful, we hope a community will help maintain it. For this reason, we have tried to make sure it easy to extend IDE++ to new plugins and views, as described in Section 4.4.

3.2 The Architecture of IDE++

At its core, IDE++ realizes the publish and subscribe model on two levels. In the context of Eclipse, its host IDE, IDE++ is itself a subscriber that listens for events coming into the IDE and the IDE's responses to those events. It is with respect to these events that IDE++ strives to be comprehensive and fine-grained. IDE++ then republishes these events to the ecosystem of user-aware applications that it enables; from the perspective of its clients, IDE++ is a publisher to which they subscribe. IDE++ collects IDE interactions in three ways: For view switching, IDE++ extends Mylyn Monitor by refining its capture of selections; to capture edits at the granularity of keystrokes, IDE++ directly instruments the Eclipse editor; for code completion and refactoring, it augments Eclipse's default interaction collection facilities.

The Mylyn Monitor, which focuses on supporting task-oriented workflows, monitors all view-switch interactions (exposed in `IPartListener` in Eclipse) and we directly used their listener `MonitorUi` for this purpose. However, for user selection interactions, the Mylyn Monitor publishes only one event for all kinds of selections. To get fine-grained selection interactions, IDE++ refactored the Mylyn Monitor's `MonitorUi` by splitting its methods into sets of methods. As an example, IDE++ specialized the `handleWorkbenchPartSelection` method into three methods: `structuredSelection`, `textSelection`, and the `otherSelection` method, each handling different selection content. When capturing an interaction, IDE++ also extracts associated information, such as the name of an opened view.

The most challenging interactions to capture are editing interactions. While the Mylyn Monitor does capture editing events, it does not do so at the granularity we seek. We could not find the programmer's keystrokes in Mylyn's edit event and, for a long editing session without switching perspectives, Mylyn generated only a single edit event. Thus, we turned to Eclipse's Java editor. The Eclipse JDT plugin manages Java editor events in a decentralized way: it provides listeners in different modules to capture mixed of fine- and coarse-grained change notifications. For example, `IElementChangedListener` publishes AST changes and `IDocumentListener` notifies file buffer changes.

IDE++ aims to provide more consistent, meaningful and fine-grained change types including the low-level change's to the editor's buffer and high-level language semantic changes such as refactoring. To capture fine grained text editing interactions, we implemented two listeners. First, we implemented `IFileBufferListener` to publish the opening or closing of a text file buffer. Once we know a text file buffer has been opened, we attach an `IDocumentListener` to its backing document to capture `DocumentEvent` instances. For each change, a document event contains its offset, its length, and, if it is an insertion, its text. Eclipse's support for attaching a listener to refactoring interactions is poor. For instance, when a user performs a copy or rename refactor operation, a refactoring event is fired to `RefactoringExecutionEvent` listeners. However, this event does not give enough information to fully resolve the changes that will be performed. To solve this problem, we built modules that participate in the refactoring process to distinguish the changes that refactoring eventually makes from its inputs.

Eclipse provides hooks to allow developers to register interaction monitors; users need to implement the exposed listeners. To monitor how users change a class in Java development, IDE++ implements `IElementChangedListener`, which publishes changes to a class, such as adding a field. Unfortunately, some of these listeners do not comprehensively capture their entity's interactions. For example, `IElementChangedListener` does not publish the renaming of a field. For some events, the monitoring process in Eclipse is centralized: it provides one listener for all the events in different views, which sometimes conflates the events. `CommandMonitor` is an example; it monitors all the command events from all the views. We refactored `CommandMonitor` into a set of listeners, one for each view. For example, we created `DebugCommandListener` to capture all commands in the debug view: resume, suspend, step into, *etc.*

IDE++ extensively instruments Eclipse to intercept interactions. Generally, instrumenting programs slows them down, and often imposes an unacceptable performance penalty. Programming is inherently interactive; it interleaves many tasks, such as thinking, editing, and navigating. Thus, the instrumentation IDE++ adds is unusual in that it is confined to IO with a human. Humans are glacially slow compared to computers; relative to human reaction time, which averages 190ms [35], IDE++'s overhead is negligible.

```
2012-03-28 18:16:56,090|main|35|95|p
2012-03-28 18:16:56,650|main|35|96|u
2012-03-28 18:16:56,762|main|35|97|b
2012-03-28 18:17:01,002|main|36|97|1
2012-03-28 18:17:01,711|main|36|96|1
2012-03-28 18:17:01,961|main|35|96|r
2012-03-28 18:17:01,995|main|35|97|i
2012-03-28 18:17:02,137|main|35|98|v
2012-03-28 18:17:02,153|main|35|99|a
2012-03-28 18:17:02,165|main|35|100|t
2012-03-28 18:17:02,170|main|35|101|e
2012-03-28 18:17:04,172|main|28|Save
```

Figure 4: Sample interaction log recorded by IDE++

3.3 Interaction History

The syntax of a single interaction captured by IDE++ contains four fields: time stamp, thread id, type, and content. The type field, recorded as an integer, identifies an interaction, such as a view-selection or a edit. Currently, IDE++ supports 44 kinds of interactions, documented at `http://marketplace.eclipse.org/content/ide#.UzGgj1FdV3c`. Some interactions have associated content. For instance, an editing interaction includes the characters that have been typed; these characters are stored in the content field.

Figure 4 shows an example of interactions captured by IDE++. Type 35 denotes keystroke and type 36 denotes a Backspace keystroke. The numbers (95-101) after the type information of the interactions record the offset of the edits in the file. The characters that were typed follow the offset. The last interaction (type 28) tells that the user performed "Save" command. The sample log records the following actions performed by a user: He begins to create a public field. Then he decides to change it to private. So he removes "ub", types "rivate" and clicks "Save".

IDE++ interaction logs grows linearly in the number interactions between a programmer and her IDE. As noted above, IDE++ captures these interactions on the very slow path to a human. We exploit this fact to filter events online that would otherwise be prohibitively expensive and relegated to postprocessing. For this reason, IDE++ only republishes events with a registered listener.

3.4 Subscribing to IDE++ Events

IDE++ supports both online and offline analysis of interaction information. The online analysis is a prerequisite for building "smart" IDEs that know what a programmer is doing and offer live assistance. It will form the basis of an ecosystem of user-aware IDE applications. The offline analysis allows retrospective analysis a programmer's interactions. The IDE++

```
notifyAdded( IJavaElement element );
notifyCodeChanged( IJavaElement element );
notifyCopied(
  IJavaElement element, IJavaElement from,
  IJavaElement to
);
notifyMoved( IJavaElement from, IJavaElement to );
notifyRemoved( IJavaElement element );
notifyRenamed( IJavaElement from, IJavaElement to
     );
notifySignatureChanged( IJavaElement element );
notifySuperTypesChanged( IType type );
```

Figure 5: API methods in `JavaModelListener`.

infrastructure enables any plugin to subscribe to its interaction information. Internally, IDE++ sets up monitors when Eclipse launches and receives events while Eclipse remains open. It provides a set of listener APIs to which applications can subscribe to receive interactions.

Currently, IDE++ offers six listener interfaces. The first listener, `DebugBreakpointListener`, captures breakpoint interactions; `DocumentChangesListener` captures changes in documents; `JavaLaunchListener` captures program launch information; `JavaModelListener` captures editing interactions; `JUnitListener` captures JUnit interactions; and, the final listener, `UserActivityListener`, captures UI and command interactions. Figure 5 shows the API methods in `JavaModelListener`. Developers only need to implement listeners that capture the interactions they want. To lower the burden on developers, IDE++ provides adapters with do-nothing implementations of the listener interfaces. These adapters allow developers to focus on their application's logic instead of littering their code with irrelevant methods.

To persist a programmer's interactions, IDE++ leverages its own framework of listeners: The log file is produced by a meta-listener that implements and registers with all of IDE++'s listeners. This listener then echos incoming events into the log file. There are two main concerns regarding log files: 1) log files might become very large and 2) programmers do not want sensitive information such as source code and author information to leak to unauthorized applications. The measures IDE++ takes to handle log size are addressed above in Section 3.3. IDE++ is designed to support local IDE plugins. Thus, IDE++ handles privacy concerns in the same way Microsoft Excel does — *viz.* share nothing by default and instead leave the management of the log files to the discretion of the user. In addition, users can choose to hash the concrete information such as the source code edits to prevent it from being leaked.

To help developers build plugins, we have published documentation, tutorials and examples showing how to use the IDE++ infrastructure at `http://marketplace.eclipse.org/content/ide#.UzGgj1FdV3c`.

3.5 Extending IDE++

The set of IDE interactions is a moving target. New plugins will introduce new interactions. IDE++ has an open design that allows integration of new interactions easily. Integrating a new interaction requires two steps: 1) extending the subscriber to monitor the new plugin to get change notifications and 2) adding a new listener in the publisher side to allow clients to retrieve interactions. We illustrate the two steps using the EGit plugin as an example.

To subscribe to change notifications from a new plugin, we need to find out the listeners it provides. EGit provides an `IndexChangedListener` that is notified when the Git index changes. IDE++ subscribes to this listener to receive notifications from EGit and then republishes them to the IDE++ listeners.

The remaining step is to allow other applications to receive the new interactions from the new publisher. First, we need to parse a plugin's notification object to extract its information. For EGit, we retrieved the `Repository` object from the notification. We then call the relevant `notifyIndexChanged` method on the IDE++ listeners and pass the repository as a parameter. Other applications can now implement the listener and register it with IDE++ to access to this piece of the interaction stream.

The subscriber and publisher architecture makes monitoring and exploiting interactions straightforward. Often, IDE++ needs only to subscribe to a plugin's interactions then republish them without modification to other applications. When this is not the case, a developer who is familiar with the plugin should find it easy to export its interactions to IDE++. Finally, extending a new plugin is a one time task that opens the door to IDE++'s ecosystem of user-aware applications.

4. Evaluation

Our evaluation objective is two-fold: to demonstrate that IDE++ comprehensively captures fine-grained IDE interactions, and to show the promise of that information as the basis of an ecosystem of user-aware IDE applications.

4.1 Comprehensiveness and Granularity

To be the basis of a vibrant ecosystem of user-aware applications, IDE++ must effectively realize the goal of comprehensive and fine-grained interaction capture. Here, we present two experiments that measure the degree to which we succeeded. The first experiment shows that we are able to fully replay nontrivial sequences of IDE interaction from IDE++'s interaction history. The second experiment quantifies IDE++'s event capture against Mylyn Monitor.

4.2 IDE Interaction Replay

In this experiment, IDE++ records the actions of Programmer A as he performs programming tasks. Then we show that, given the same initial environment as A and using A's interaction log, programmer B can redo exactly what A did

			Edit		Browse		Test		Debug		Total	
	Task	**Time (min)**	IDE++	Mylyn	IDE++	Mylyn	IDE++	Mylyn	IDE++	Mylyn	IDE++	Mylyn
RPNCalculator		32	1,367	12	98	98	25	2	39	28	1,529	140
Convert E to C	*String*	23	888	10	142	142	12	1	9	1	1,051	154
	Repeat-Until	44	3,211	26	144	144	64	3	23	7	3,442	180
	Array	94	8,233	47	246	246	81	3	24	13	8,584	309
	Boundary	25	720	8	165	165	40	2	11	3	936	178
	Every	40	3,020	25	228	228	5	1	3	1	3,256	255

Table 2: A comparison of the interactions captured by IDE++ and Mylyn Monitor.

	Task	**Participant**	**Time (min)**
RPNCalculator		Student A	21
Convert E to C	*String*	Student A	15
	Repeat-Until	Student B	33
	Array	Student B	61
	Boundary	Student C	21
	Every	Student C	33

Table 3: Time used for participants to replay the interactions for each task.

and produce the same output. We conducted this user-session replay experiment to demonstrate how well we realized our goal of achieving the systematic capture of maximally fine-grained events. This experiment rests on the intuitive idea that, if one can precisely replay an IDE interaction from an IDE++ log, then we have indeed met our goal.

Table 2 lists the six programming tasks we used in this experiment. The `RPNCalculator` task requires a programmer to write a reverse polish notation calculator and provide JUnit test cases to ensure correctness. An undergraduate course in our department assigned programming tasks that involved adding support for syntactic constructions to a pedagogical language E by translating them into C. The five constructs were *String*, *Repeat-Until*, *Array*, Array *Boundary* check, and *Every*, a loop construct similar to a `foreach`. One of the authors completed these tasks while IDE++ recorded his interactions. The second column records the time he used to finish each task.

IDE++'s raw output is not easy for humans to read, so we processed it to separate user actions from Eclipse's responses and to map file offsets into a line number and column. Figure 6 shows postprocessed output. Participants in the replay experiment simply follow the logged user actions[1].

We invited three students to participate in the experiment. All of them had moderate coding experience and were familiar with the Eclipse IDE. Each participant performed the replay experiment for two of the six assignments in Table 2. Table 3 shows the time taken to replay the interactions for

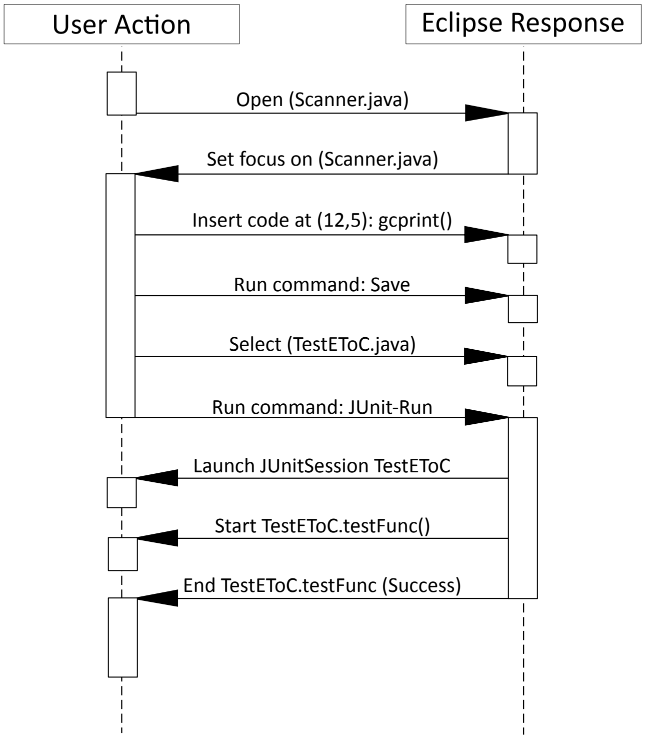

User Action
Eclipse Response
Open (Scanner.java)
Set focus on (Scanner.java)
Insert code at (12,5): gcprint()
Run command: Save
Select (TestEToC.java)
Run command: JUnit-Run
Launch JUnitSession TestEToC
Start TestEToC.testFunc()
End TestEToC.testFunc (Success)

Figure 6: Sample natural language interaction sequence diagram.

each programming task. We diffed their source code files against the target files and confirmed that they match. To make sure the entire process was replayed, we also compared the logs produced by the participants with the author's logs and confirmed that the interactions recorded are the same with the exception of the time stamps.

4.3 Mylyn Monitor Comparison

Like IDE++, Kersten's and Murphy's Mylyn Monitor captures IDE interactions; its focus is capturing those interactions needed to understand and support task-oriented workflows. In contrast, IDE++ seeks to be a general purpose framework for a new class of user-aware IDE applications. Although the two projects differ in focus, Mylyn Monitor is a mature,

[1] Automating replay is future work.

well-engineered project. Thus, we use it as a baseline against which to understand the detail of IDE++'s interaction capture.

To enable the comparison of interaction logs between IDE++ and the Mylyn Monitor, we normalized both project's interactions into atomic actions performed by a user, such as a mouse click. We grouped the actions into four categories: editing, browsing, testing, and debugging; we then compared the number of actions recorded for each category.

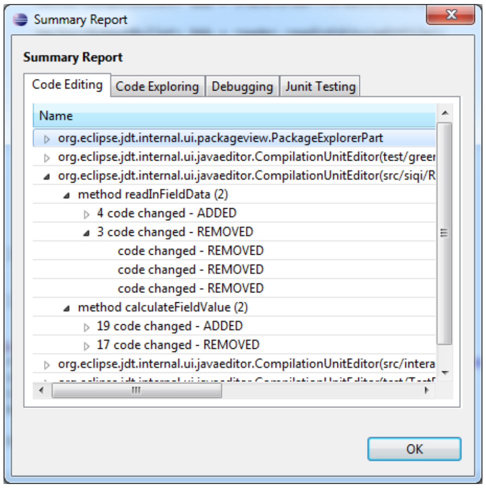
Table 2 shows the number of actions captured by both monitors for each category. While Mylyn Monitor has been extend to capture screen contents [5], it is clear that IDE++ captures far more editing interactions. This is because IDE++ captures all of the fine-grained interactions including cursor movement, keystrokes, and related commands, while the Mylyn Monitor records only coarse-grained file change events and commands. Browsing actions include selecting structured content and switching views. The Mylyn Monitor was designed for this task; IDE++ also captures these actions and more. For the testing category, the Mylyn Monitor records only that the Run-Test command was performed, while IDE++ also includes which test cases have been run and their results (success or failure). For debugging, IDE++ records when a user enables, disables, or changes a breakpoint, the debugging commands a user uses, such as Step-Into and Step-Over, which variables he has inspected while his program was paused, and the stack frames he selected. The Mylyn Monitor records only the commands run and that a variable or stack frame was selected, but no data about it.

4.4 Developer-aware IDE Applications

To demonstrate the necessity of comprehensive and fine-grained interaction information and the promise of an ecosystem of user-aware applications built on IDE++, we introduce, in addition to Sage (introduced in Section 2), three IDE++ applications: DevTime, Proctor, and Localizer. These applications help programmers edit, test, and debug. Figure 11 in Section 4.5 shows the source code of a simple, yet meaningful, application to illustrate how easy it is to write an application using IDE++.

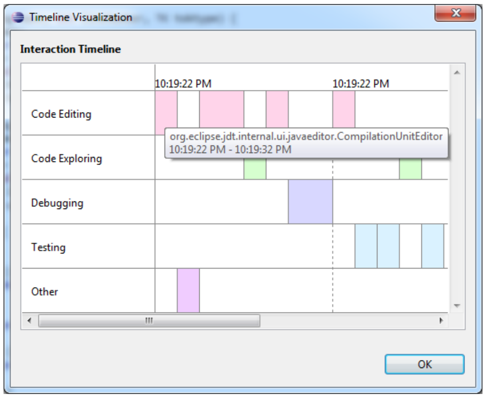
4.4.1 DevTime

After finishing a task, a programmer may want to review what he did to track a project's progress or file a daily working report. Typically, he would review those changes in his version control system (VCS). However, VCS history is a coarse record of what he actually did: it does not reflect the time he spent browsing code or running regression tests. If he made several changes in a single location to the source in his editor, VCS can capture only those changes actually committed to its history. DevTime has two reports: a summary of task performed by category, shown in Figure 7, and a timeline visualization in Figure 8.



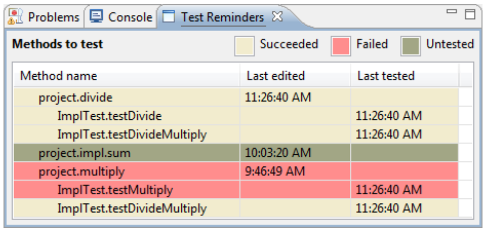
```
Name
▷ org.eclipse.jdt.internal.ui.packageview.PackageExplorerPart
▷ org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor(test/greer
◢ org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor(src/siqi/R
    ◢ method readInFieldData (2)
        ▷ 4 code changed - ADDED
        ◢ 3 code changed - REMOVED
              code changed - REMOVED
              code changed - REMOVED
              code changed - REMOVED
    ◢ method calculateFieldValue (2)
        ▷ 19 code changed - ADDED
        ▷ 17 code changed - REMOVED
▷ org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor(src/intera
```

Figure 7: The Summary Report application shows a programmer how he has interacted with Eclipse.



org.eclipse.jdt.internal.ui.javaeditor.CompilationUnitEditor
10:19:22 PM - 10:19:32 PM

Figure 8: IDE++ draws a timeline of a user's interactions for the most recent session.



Methods to test ☐ Succeeded 🟥 Failed ▨ Untested

Method name	Last edited	Last tested
project.divide	11:26:40 AM	
ImplTest.testDivide		11:26:40 AM
ImplTest.testDivideMultiply		11:26:40 AM
project.impl.sum	10:03:20 AM	
project.multiply	9:46:49 AM	
ImplTest.testMultiply		11:26:40 AM
ImplTest.testDivideMultiply		11:26:40 AM

Figure 9: Proctor tracks editing and testing interactions to remind the user which methods have not been tested yet.

4.4.2 Proctor

A good software engineering practice is to test a method while the method and changes made to it are still fresh in a programmer's mind. The longer the gap between editing and testing, the harder it is for a programmer to find and fix a bug he introduced. When shifting from composing code to testing, a programmer must recall the methods he edited and whether he already tested them to best direct his testing efforts. The IDE will save him time when making this transition if it tracked which methods have been edited and tested recently.

Proctor helps programmers build testing plans by tracking which methods have been edited and tested recently. It monitors the editing interactions to get the list of methods that have been changed and JUnit test interactions to get the list of run test cases. By scanning the source code of the test methods, the Proctor knows which methods have been tested and whether they passed the test or not. It organizes the edited methods into three categories and presents them to the programmer: methods that have not been tested yet, methods that have been tested and passed, and methods that have been tested but failed. Figure 9 shows example output.

4.4.3 Localizer

Bug localization is an active research area, comprising statistical models, code history, program slicing, *etc.* [14, 15, 20, 21, 37]. Many techniques apply sophisticated analysis to an entire program. The IDE++ Localizer introduces a new approach: localizing bugs by searching the recent editing history. Programmers run regression test cases routinely to ensure that recent edits have not adversely affected existing functionality. When a test case fails, it is likely that a recent edit caused the failure. Using this heuristic and the program's call graph, the IDE++ Localizer lists recently edited methods that might have caused a JUnit test failure. Since this approach requires only recent editing history, it is light-weight and provides live feedback.

Consider the example shown in Figure 10. A programmer changed both the `sum` and `sayHello` methods in `Calculator` during the current session. When he ran the test cases, `testSum` failed. Although both `sum` and `sayHello` were changed, since the call graph of `testSum` shows that only `sum` affects it, the Localizer tells the programmer that `sum` is the candidate method that caused the failure.

Localizer illustrates the use of a particular kind of interaction; it monitors only the JUnit launch and editing interactions, and is triggered by a test failure. First, it builds the set of the methods called by the test method. Then it extracts the set of methods edited during the recent sessions from IDE++'s interaction history. The intersection of these two sets forms the set of candidates. Finally, Localizer presents these candidates to the programmer. Since the edits triggering the error might not have occurred in the most recent session, Localizer can search the edit history of previous sessions. By default, Localizer searches the last three sessions. The programmer can override this default.

4.5 Writing IDE++ Applications

Table 4 displays the lines of code (LOC) of the four applications we built. The third column lists the total LOC; second column shows the LOC related to receiving and processing IDE interactions from IDE++. DevTime retrieves interactions directly from IDE++ log files, so its LOC for interactions is only 20. Because Sage monitors every available interaction and parses the arguments to get information, it requires the most logic to handle interactions. Additionally, it contains many different automata that check for different patterns, explaining why it is much larger than the other three plugins. Comparing the second and third columns, we see that developers need to write very little code to retrieve and utilize interaction information from IDE++.

	LOC	
Application	**Interaction**	**Total**
DevTime	20	972
Sage	278	2,123
Proctor	69	889
Localizer	130	874

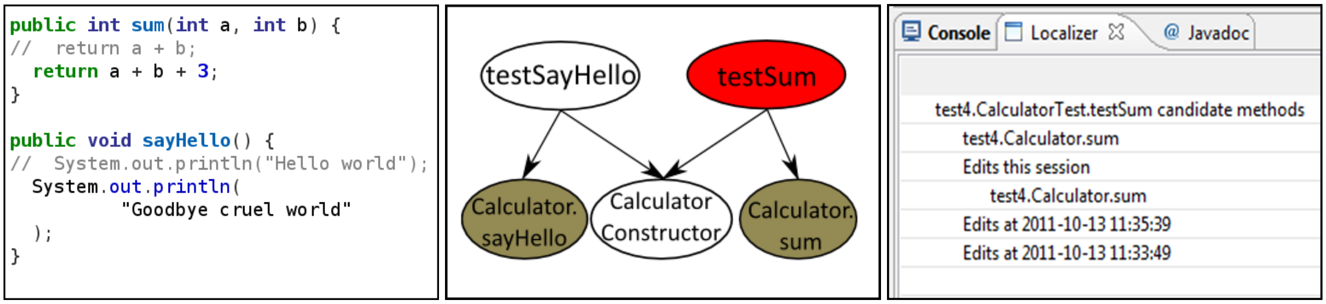
Table 4: Line of code (LOC) of the four applications; the Interaction column displays the LOC dedicated to receiving and processing IDE interactions from IDE++; the different between this column and the Total column is the application-specific logic.

We use a simple, albeit real, example to illustrate how easy it is to build an IDE++ application. Assume a developer wants to track the average running time of the programs run during an Eclipse session. For this application, a developer needs only implement and register `JavaLaunchListener` to intercept program launch interactions. In Figure 11, the developer puts the bulk of the time-averaging logic in `handleTerminated` and stores the results in `runningAverages`. Despite its brevity, the code demonstrates a complete use of the IDE++ infrastructure.

5. Related Work

Many IDEs, Eclipse among them, support rudimentary user monitoring [30]. They provide hooks that allow developers to implement their own listeners. However, the support for capturing interactions is ad hoc and cumbersome. For example, Eclipse only partially captures UI interactions, since it does not capture mouse actions; when a programmer issues a command, Eclipse does not report whether the programmer typed a hotkey or clicked a button in the tool bar or a menu. In contrast, IDE++ comprehensively captures and republishes IDE interactions in a standard, easily parsed format.

Code evolution dominates the software life cycle. Developers use a version control system (VCS) to track code evolution. To support collaborative development, VCS allows users to write and commit code to a shared repository. IDE++ also tracks code evolution, but at a finer granularity: we capture every edit interaction, as the changes in an buffer between two idle periods of at least one second. When a developer is



```
public int sum(int a, int b) {
//  return a + b;
  return a + b + 3;
}

public void sayHello() {
//  System.out.println("Hello world");
  System.out.println(
            "Goodbye cruel world"
  );
}
```

test4.CalculatorTest.testSum candidate methods
test4.Calculator.sum
Edits this session
test4.Calculator.sum
Edits at 2011-10-13 11:35:39
Edits at 2011-10-13 11:33:49

(a) Code change. (b) Call Graph of test case methods. (c) The Localizer result.

Figure 10: How Localizer works: (a) A programmer changes both `sayHello` and `sum` (marked gray in (b)). (b) `testSum` fails when the test cases were run (marked red). (c) Localizer suggests that changes in `sum` might have caused the failure.

```
class RunningAverage {
  int count;
  double average;
  RunningAverage(double a) {
    count = 1; average = a;
  }
}
JavaLaunchListener l = new JavaLaunchListener() {
  private Map<IType, Long> launched =
    new HashMap<IType, Long>();
  private Map<IType, Long> runningAverages =
    new HashMap<IType, RunningAverage>();
  public void programLaunched( IType mainType ) {
    launched.put(
      mainType, System.currentTimeMillis()
    );
  }
  public void programTerminated( IType mainType ) {
    Long started = launched.remove( mainType );
    if (started != null) {
      long runTime =
        System.currentTimeMillis() - started;
      RunningAverage rAve =
            runningAverages.get(mainType);
      if (rAve == null) {
        runningAverages.put(
          mainType, new RunningAverage( runTime );
        );
      }
      else {
        rAve.average *= rAve.count;
        rAve.average += runTime;
        rAve.average /= ++rAve.count;
      }
    }
  }
};
IDEPPPlugin.addListener(l);
```

Figure 11: Instrumenting program launch interactions.

editing, he might make several changes at the same location in the source before committing it. IDE++ captures all of the edits while a VCS captures only the difference between commits. By capturing these granular edits instead of file saves, IDE++ comes closer to capturing a developer's thought process. For example, a tricky problem might cause a developer to navigate back and forth between files, make and unmake a change, before reaching a decision.

Kersten's and Murphy's Mylyn Monitor was an important step in the capture of IDE interactions [16]. Indeed, many recent IDE applications, which we discuss next, depend on the Mylyn Monitor. IDE++ continues the Mylyn Monitor's pioneering work along three dimensions — comprehensiveness, ease-of-use and granularity. IDE++ seeks to intercept all IDE interactions, as identified by the phases of the software life cycle. The set of IDE interactions is a moving target, so at any instance in time, especially when a new plugin gains traction, IDE++ will fall short of this goal. Thus, IDE++ has been designed to make it easy for programmers to extend it to new classes of interactions.

Robbes and Lanza proposed Spyware an IDE monitor that captures fine-grained editing interactions [26]. Like IDE++, Spyware is a framework on which to build applications. Unlike IDE++, Spyware exclusively intercepts edits, ignoring other IDE interactions. Vakilian *et al.*'s CodingSpectator and CodingTracker aim to capturing low-level code refactoring changes [32]. Yoon *et al.* present Fluorite that captures low-level editing interactions [36]. The above work all focus on a subset of interactions, while our work advocates the systematic monitoring all kinds of interactions.

Applications Program comprehension is an important part of the software engineering process. Researchers have applied interaction information to aid program comprehension. Fritz *et al.* proposed a model using interactions captured by the Mylyn Monitor to judge a programmer's knowledge of code [9]. Kersten and Murphy use the Mylyn Monitor to produce a recommender for the next method to edit using their degree of interest (DOI) measure, which is derived from a database of interaction traces [16]. Guzzi *et al.* proposed a new type of interaction, collective code bookmark to summarize source code to help programmers understand software artifacts [1]. Guzzi *et al.* also presented a micro-blogging technique: group a series of interactions and attach a message to describe the interactions to enhance program comprehension [10]. Ko *et al.* studied the relationship between interactive aspects of

IDEs and program understanding [18]. IDE++ can complement these applications by providing more information in the form of finer-grained interactions and additional classes of interactions, such a debugging interactions. For example, the sequence of debugging commands a programmer issues might indicate his degree of knowledge of some code.

A large body of work on in-program, assistative agents for general purpose applications exists. The Lumière project, which culminated in the Office Assistant in Microsoft Office, is one notable example [13]; the fact that Office Assistant became derisively known as Clippy and publicly "retired" by a Microsoft's CEO in front of a cheering audience[2], is a testament to the importance of default settings and deployment, not the promise of the underlying technology. Lumière's focus is Bayesian learning, but also internally abstracts its event stream to explicitly represent repetition and inter-event gaps, with which we intend to experiment. A more recent example is Ekstrand *et al.*'s work, where the researcher combines free-form text query with in-program context sensitivity to improve search results [7]. IDE++ can be seen as the specialization of this line of work to the IDE domain; We believe that the IDE++ ecosystem of user-aware applications will quickly grow to encompass those that apply Machine Learning to its stream of IDE interactions.

Brun *et al.* proposed speculative analysis that leverages idle multicores to anticipate what a user may next wish to do, such as compile or run JUnit, and kick off these tasks in the background, shifting them to the foreground if the guess is correct [3]. For instance, they speculatively apply Eclipse's quick fix tips in the background, then tell users which ones worked. For version control interactions, they speculatively merge a developer's current branch in the background to report how many conflicts would arise [4]. IDE++ allows the extension of speculation to other programmer interactions. For example, by monitoring editing interactions, the IDE could run relevant test methods in the background, and notify the programmer about failures.

Researchers have also employed interaction history for prediction. Robbes *et al.* used Mylyn Monitor interaction logs to improve program change prediction [29]. Robbes and Lanza used edit history to improve IDE code completion [25]. Lee *et al.* described a set of micro interaction metrics, such as how much time a programmer spends in one file and how many selection operations he makes to predict bugs [19]. Purandare *et al.* present a general framework for optimizing the monitoring of loops [24]. IDE++'s fine-grained interaction information provides more data as input to predictors. As an example, Lee's bug prediction model could include editing interactions: A file that has been changed many times is likely to contain bugs. IDE++ allows the construction of prediction models for new classes of development activities, such as running a test.

6. Conclusion and Future Work

The interactions between programmers and their IDEs contain valuable information, much of which currently goes to waste. Systematically recorded into an easy-to-use format, these interactions could usher in new, highly personalized, user-aware applications with the potential to improve programming productivity. In this paper, we have introduced IDE++, an IDE interaction monitor, to set the stage for interpreting and exploiting these interactions, tasks that require their own research agenda. We have built four applications — DevTime, Sage, Proctor, and Localizer — to demonstrate the promise and utility of the new ecosystem of applications IDE++ makes possible. We have published these applications as well as the IDE++ infrastructure onto the Eclipse Marketplace. We welcome users to try it.

Programmer interaction histories are good candidates for using data mining techniques to discover previously unknown patterns. Experienced programmers' interactions facilitate knowledge reuse and provide new educational opportunities. Of course, sharing the interaction information raises privacy concerns. We intend to apply existing techniques, such as CQual [8], taint analysis [23], and sanitization [34], to IDE++ to protect contributors.

As our interaction database grows, we will use it to study questions such as, "How do the interaction histories of experienced programmers differ from those of novices?", "Do programmers from the same project share common interaction patterns?", and "Does interaction history correlate with measures such as program complexity and bug density?". We plan to extend IDE++ to support more interactions, such as support for compiler warning and collaboration plugins such as EGit and Subclipse. Our monitoring infrastructure is an open framework. We welcome other developers to build applications upon it. Tutorials, documentation, tool downloads, and updates are available at `http://marketplace.eclipse.org/content/ide#.UzGgj1FdV3c`.

Acknowledgments

This research was supported in part by NSF grant no. 1117603. The information presented here does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

References

[1] G. Anja, H. Lile, L. Michele, P. Martin, and D. Arie van. Collective code bookmarks for program comprehension. In *ICPC*, 2011.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: exploring future development states of software. In *FSE*, 2010.

[2] `http://bit.ly/pmHCwI`.

[4] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Proactive detection of collaboration conflicts. In *FSE*, 2011.

[5] B. de Alwis, G. Murphy, and M. Robillard. A comparative study of three program exploration tools. In *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, pages 103–112, June 2007.

[6] R. DeLine, M. Czerwinski, and G. G. Robertson. Easing program comprehension by sharing navigation data. In *VL/HCC*, pages 241–248, 2005.

[7] M. Ekstrand, W. Li, T. Grossman, J. Matejka, and G. Fitzmaurice. Searching for software learning resources using application context. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 195–204, New York, NY, USA, 2011. ACM.

[8] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *PLDI*, 1999.

[9] T. Fritz, G. Murphy, and E. Hill. Does a programmer's activity indicate knowledge of code? In *FSE*, 2007.

[10] A. Guzzi, M. Pinzger, and A. van Deursen. Combining Micro-Blogging and IDE interactions to support developers in their quests. In *ICSM*, 2010.

[11] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, Mar. 2009.

[12] J. S. Hammond. IDE usage trends, 2008.

[13] E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The lumière project: Bayesian user modeling for inferring the goals and needs of software users. In *In Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence*, pages 256–265. Morgan Kaufmann, 1998.

[14] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE*, 2002.

[15] M. Kamkar, N. Shahmehri, and P. Fritzson. Bug localization by algorithmic debugging and program slicing. In *PLILP*, 1990.

[16] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *FSE*, 2006.

[17] R. B. Kline and A. Seffah. Evaluation of integrated software development environments: Challenges and results from three empirical studies. *Int. J. Hum.-Comput. Stud.*, 63(6):607–27, Dec. 2005.

[18] A. Ko, B. Myers, M. Coblenz, and H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE TSE*, 2006.

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

[20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *PLDI*, 2003.

[21] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *FSE*, 2005.

[22] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Elipse IDE? *IEEE TSE*, 2006.

[23] J. Newsome and D. X. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS*, 2005.

[24] R. Purandare, M. B. Dwyer, and S. Elbaum. Monitor optimization via stutter-equivalent loop transformation. In *OOPSLA*, 2010.

[25] R. Robbes and M. Lanza. How program history can improve code completion. In *ASE*, 2008.

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

[27] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How Do Professional Developers Comprehend Software? In *ICSE*, pages 255–265. IEEE, 2012.

[28] F. Roethlisberger. *Management and the Worker*. Harvard University Press, 1939.

[29] R. Romain, P. Damien, and L. Michele. Replaying IDE interactions to evaluate and improve change prediction approaches. In *MSR*, 2010.

[30] The Eclipse Foundation. Eclipse instrumentation framework. http://dev.eclipse.org/viewcvs/viewvc.cgi/platform-ui-home/instrumentation/index .html?revision=1.12.

[31] The Economist. The quantified self: Counting every moment. *The Economist Magazine*, 2012.

[32] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, R. Zilouchian Moghaddam, and R. E. Johnson. The need for richer refactoring usage data. In *EUPLT*, 2011.

[33] P. Viriyakattiyaporn and G. C. Murphy. Improving program navigation with an active help system. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pages 27–41. IBM Corp., 2010.

[34] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *PLDI*, 2007.

[35] Wikipedia. Mental chronometry, Visited March 2014. `http://en.wikipedia.org/wiki/Mental_chronometry`.

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

[37] A. Zeller. Yesterday, my program worked. today, it does not. why? In *FSE*, 1999.