How Are Java Software Developers Using the Eclipse IDE?

Gail C. Murphy, Mik Kersten, and Leah Findlater, *University of British Columbia*

The Eclipse integrated development environment continues to gain popularity among Java developers, but are they taking full advantage of its features and third-party plug-ins?

Many software developers spend their workday in an integrated development environment. For many Java developers, Eclipse is the IDE of choice.[1,2] Commonly cited reasons for using Eclipse include rich Java Development Tools (JDT) support and a plug-in architecture that allows tight integration of third-party functionality. But are developers using these features and plug-ins? To help answer these questions, we report on usage data collected from 41 Java software developers using Eclipse, providing a glimpse into their work habits.

The usage data

Ninety-nine software developers agreed to participate in our user study of Mylar, an Eclipse plug-in we're developing (see www.eclipse.org/mylar). These developers represent early adopters willing to run on the most recent releases of Eclipse and take advantage of its extensibility. Table 1 presents the developers' job functions and industry sectors. The developers were using Eclipse 3.1 and milestone releases of Eclipse 3.2, specifically M1 and M2. Of these 99 developers, 74 submitted usage data.

In analyzing the usage data, which we collected in the latter half of 2005, we focused on JDT use, not on Mylar. The data consists of traces of each developer's interactions with Eclipse recorded by our Mylar Monitor plug-in (see the related sidebar). Each trace, called an *interaction history*, includes information about the user's view and editor selections, the commands invoked, and changes made to the preferences and perspectives (a perspective is a package of various views). The information for each event includes the time invoked, the kind of event, the plug-in of which it was a part, and a string handle of the targeted event (which was obfuscated when the data was reported to ensure confidentiality of the systems on which the developers worked).

Periodically, Mylar would prompt the developer to upload his or her interaction history in XML. Figure 1 shows a sample selection and command captured in an interaction history. The selection event reflects a developer selecting a part of a package in the Pack-

0740-7459/06/$20.00 © 2006 IEEE

Table 1

Background of the developers who took part in the study

Job	Percentage
Application developer	65
Academic	13
Application architect	12
Manager/CIO/CTO	4
Other	6
Organization size	
One individual	19
Fewer than 50 employees	32
50 to 500 employees	26
More than 500 employees	23
Sector	
Software manufacturing	48
Academic	19
Financial/retail	13
Communications/networking	7
Government	5
Other	8

age Explorer view. The subsequent command is a save of an XML file.

We processed the 74 data sets to find developers who had performed at least 5,000 selections and edits in views and editors associated with Eclipse's JDT—a number we felt represented significant JDT use. We report here on the 41 developers who met this criterion. On average, the interaction histories we analyzed contained 65,492 events per developer, with a minimum of more than 11,000 events and a maximum of more than 200,000 events (standard deviation ± 50,391 events). These interaction histories suggest that developers, on average, actively used Eclipse for 66 hours, with a minimum of 10 hours and a maximum of 172 hours (standard deviation ± 44 hours). We collected the histories over periods ranging from six to 125 days.

We didn't constrain the configurations of the developers' Eclipse environments beyond requiring installation of the Mylar Monitor. Developers could (but didn't have to) use the Mylar task management and user-interface functionality and other third-party plug-ins when collecting interaction histories.

The Mylar Monitor

Mylar is an open source technology project hosted at www.eclipse.org/mylar. The Mylar Monitor is a standalone framework that collects and reports on trace information about a user's activity in Eclipse. The Mylar Monitor captures events such as preference changes, perspective changes, window events, selections, periods of inactivity, commands invoked through menus or key bindings, and URLs viewed through the embedded Eclipse browser. These events are logged to a local file and uploaded by the developer to an HTTP server. The uploading mechanism manages user IDs, provides anonymity if requested, and obfuscates the handles of targets of selections and other such user data that might be collected. This ensures that sensitive information about the source code and user isn't transmitted. Plug-ins can extend the Monitor for different kinds of user studies. The Monitor's functionality is a superset of the functionality that was available in the Eclipse Instrumentation Framework (http://dev.eclipse.org/viewcvs/index.cgi/platform-ui-home/instrumentation/index.html?rev=1.12). The key difference between the two is that the Mylar Monitor captures a full log of interaction events instead of collecting usage statistics. This allows for flexibility in analyzing usage data after it has been collected. For example, common interaction sequences and usage patterns can be determined from the Mylar Monitor's interaction event log.

A common concern about collecting trace information is the volume of information programmers can produce. We manage the size of the traces the Mylar Monitor produces through compression, using a zipped format that significantly reduces the file size. We found that the repetitiveness of user activity typically yields compression ratios over 95 percent, with a month of typical full-time programming activity resulting in an approximately 1 Mbyte trace file after compression.

To help us analyze the traces we collect, we also provide a Monitor Reports plug-in with facilities for collecting and summarizing data across one or more interaction event logs. We used this reporting framework to perform most of the processing reported in the main article.

Interaction Event
Kind: selection
Date: 2005-05-12 17:02:02.781 PST
OriginId: org.eclipse.jdt.ui.PackageExplorer
Structure Kind: java
Structure Handle: project/src/com.foo.bar/Foo.java
Interaction Event
Kind: command
Date: 2005-05-12 17:02:03.411 PST
OriginId: org.eclipse.ui.file.refresh
Structure Kind: xml
Structure Handle: com.foo.bar/build.xml

Figure 1. A stylized fragment of an interaction history for Eclipse, showing a selection and a command.

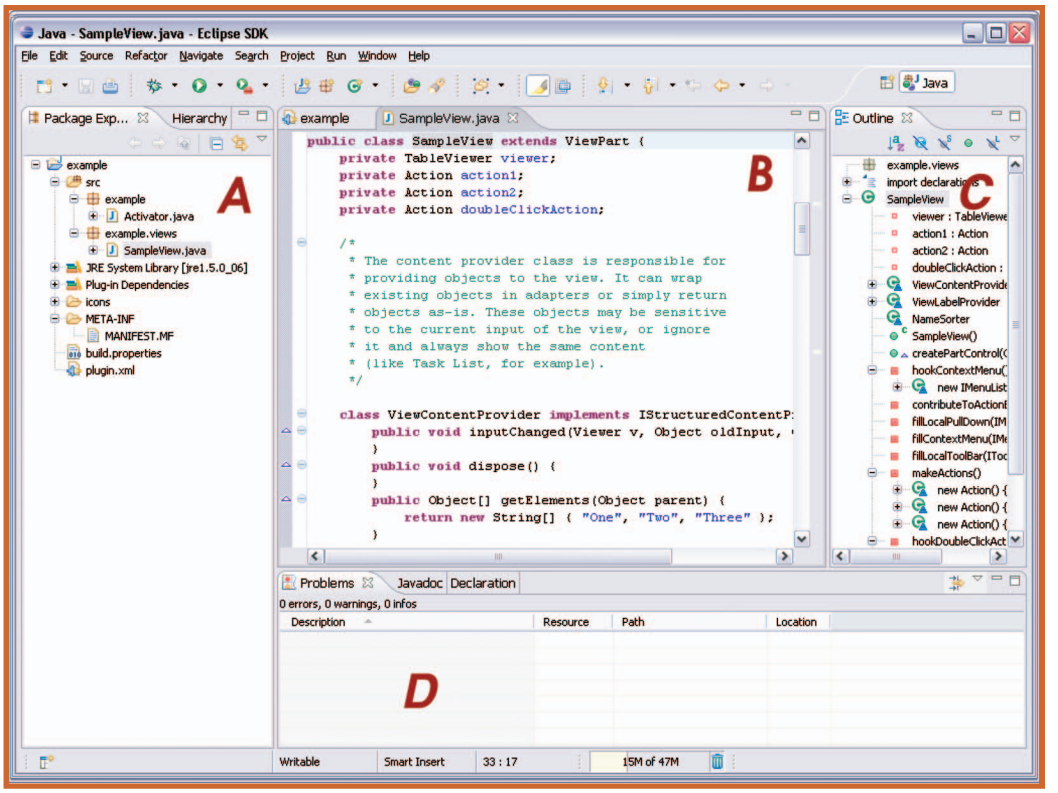


Figure 2. The default Java perspective in Eclipse: Window A is the Package Explorer view, window B is a Java editor, window C is the Outline view, and window D is the Problems view.

Use of views, editors, and perspectives

Figure 2 shows the Eclipse development environment, which lets users display and manipulate information in views. The Package Explorer view (see window A), for instance, displays information about the Java code's package, class, and member structure. The user can select information displayed in this view—such as a class—and perform a refactoring, such as renaming the class.

Window B in figure 2 shows a Java editor. In contrast to views in which an invoked command immediately affects the information, the developer must explicitly save changes in the editor. On average, 51 percent (± 8 percent, over a range of 28–71 percent) of the events in our developers' interaction histories originated in an editor—and not just a Java editor. We saw evidence of editors for more than 15 artifact types, including Ant, Antlr, AspectJ, Bugzilla, C, C#, JavaServer Pages, Perl, PhP, Python, Ruby, SQL, Tex, UML class diagrams, and XML.

Figure 3 shows the top 10 of the 42 views shipped with the default Eclipse distribution (we excluded views that weren't default views), based on the percentage of developers who made at least one selection in each of the views. The 10 views help display

- the code's static structure (Package Explorer and Outline),
- runtime and debug information (Console, Variables, Debug, and Breakpoints),
- search results (Search),
- the compiler and other tools' output (Problems), and
- general file navigation and command progress (Resource Navigator and Progress).

At least 93 percent of the developers used the first seven views.

Eclipse also supports different perspectives. Figure 2 shows, for instance, the default Java perspective with the Package Explorer, editor window, Problems view (for compiler complaints), and Outline view (for displaying the current edited file's structure). As another example, the Debug perspective makes it easy to switch to a set of views that facilitates debugging, such as a Breakpoint view, a dynamic call stack, an editor, and a Console view. The standard Eclipse JDT installation provides eight default perspectives, and users can also configure their own perspectives. Figure 4 shows the per-

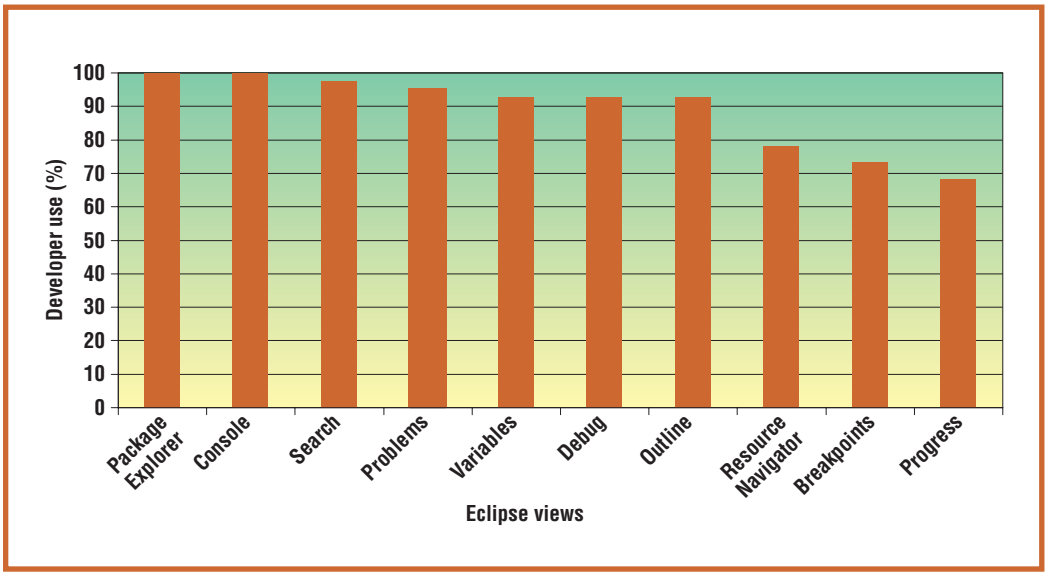


Figure 3. The top 10 of 42 views shipped with the standard Eclipse distribution, based on the percentage of the 41 developers who made at least one selection in each view.

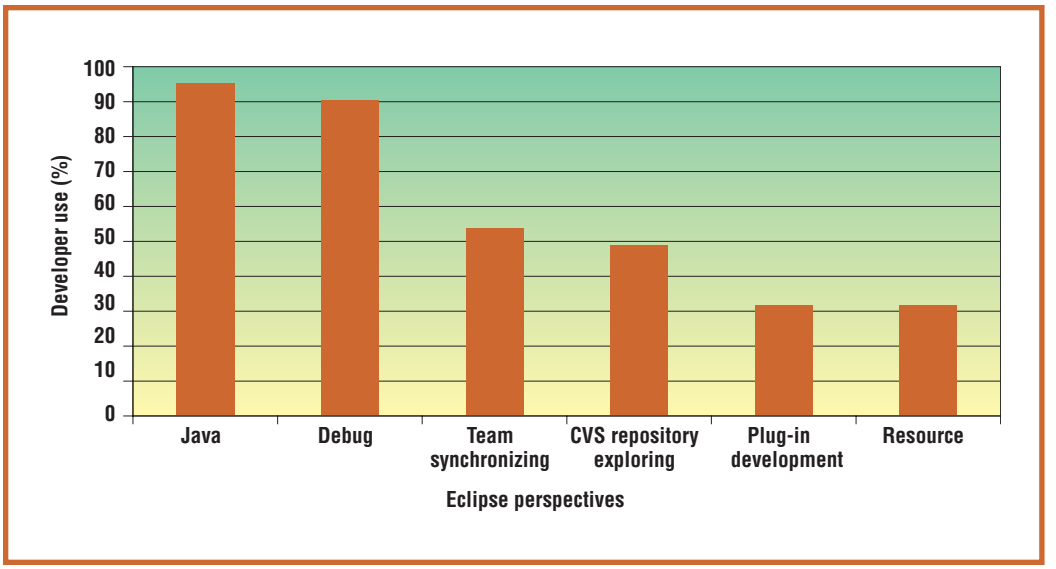


Figure 4. The perspectives (packages of views and editors) that at least 25 percent of the study's developers used.

spectives that 25 percent of the developers used at least once. The only default perspectives that none of the developers used were the Java Type Hierarchy and Java Browsing perspectives.

Frequently used commands

Across all the users, we collected data for more than 1,100 different identifiers for commands. Developers invoked a large percentage of the executed commands (84 percent) using key bindings; they invoked the remaining commands from toolbars (8 percent) and menus (5 percent), while a small number of invocation sources weren't recorded by the monitor (3 percent).

Not surprisingly, the developers used editing commands the most. Tables 2 and 3 pro-

Table 2

Top 10 commands executed by the most developers

Command	Identifier	No. of users
Delete	org.eclipse.ui.edit.delete	41
Save	org.eclipse.ui.file.save	41
Paste	org.eclipse.ui.edit.paste	41
Content assist	org.eclipse.ui.edit.text.contentAssist.proposals	41
Copy	org.eclipse.ui.edit.copy	41
Undo	org.eclipse.ui.edit.undo	41
Cut	org.eclipse.ui.edit.cut	40
Refresh	org.eclipse.ui.file.refresh	40
Show view	org.eclipse.ui.window.showViewsShortlist	40

Table 3

Top 10 commands executed across all 41 developers

Command	Identifier	Use (%)
Delete	org.eclipse.ui.edit.delete	14.3
Save	org.eclipse.ui.file.save	11.3
Next word	org.eclipse.ui.edit.text.goto.wordNext	7.3
Paste	org.eclipse.ui.edit.paste	6.8
Content assist	org.eclipse.ui.edit.text.contentAssist.proposals	6.7
Previous word	org.eclipse.ui.edit.text.goto.wordPrevious	5.9
Copy	org.eclipse.ui.edit.copy	4.6
Select previous word	org.eclipse.ui.edit.text.select.wordPrevious	3.4
Step (debug)	org.eclipse.debug.ui.debugview.toolbar.stepOver	3.2

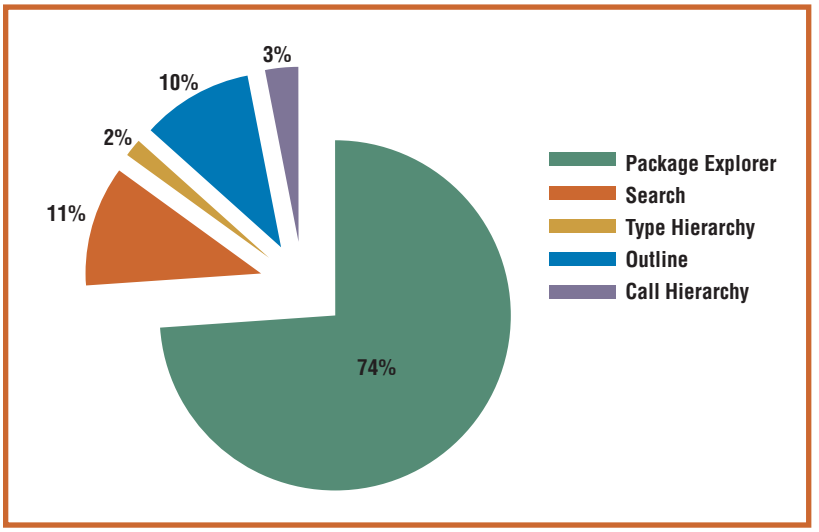


Figure 5. Use of navigation views by all 41 developers (nobody used the Declaration view).

vide two views of the top 10 commands. Table 2 lists the commands by the number of developers using the command. Table 3 lists the commands according to average use by all developers. Interestingly, developers used content assist (which suggests possible method names in the editor given a type) as much as the common editing commands.

Analyzing the command information in the interaction histories was difficult. For Eclipse and the plug-ins that extend it, the intent for the plug-in developer is to assign a unique identifier for a command regardless of how the command is made available in the environment. For instance, the same command provided through a toolbar menu and a context menu in the editor should have the same identifier. Unfortunately, not all Eclipse plug-ins use this convention. As a result, we found many inconsistencies, resulting in different identifiers representing the same command. For example, selecting Save from the File menu in the toolbar generates a different identifier than when a key binding performs the Save command. We also found cases that used the same identifier for commands provided by different plug-ins.

To account for these duplications and ambiguities, we created a mapping of identifiers that considers the context of how a command was used. This mapping reduced the number of unique identifiers from 1,208 to 1,142. However, our mapping focused on the more commonly used commands, so this number might still include duplicated and ambiguous commands. To facilitate this sort of analysis, we recommend that plug-in developers specify consistent IDs for their commands and actions.

Navigation

Most software fixes, changes, and enhancements involve navigating across the code base to understand the system's structure and the context in which code executes. Eclipse provides seven views to help a developer efficiently locate code of interest: Package Explorer, Type Hierarchy, Outline, Search, Call Hierarchy, Bookmarks, and Declaration. The developers in our study used the Package Explorer view the most, on the basis of the number of selections made in each view (see figure 5); nobody used the Declaration view, even though it is present by default in the Java perspective.

Through key bindings, Eclipse also provides direct, easily accessed support for different kinds of nonlocal navigation and searches, including navigating to the declaration of an element selected in the editor, searching for references to a selected element, and opening a type. Table 4 summarizes these nonlocal navigation and search commands available in the JDT, their key bindings on the Windows platform, how many of the developers used the commands, and each command's rank (a rank of one indicates the command that the developers used most; the lowest rank is 1,142—the number of commands). This data shows that the command used most often is opening a selected element's declaration (a rank of 21); the command used by the largest number of users is the search for references in a workspace.

To help developers mark points of interest

Table 4

Navigation and search command usage in the Java Development Tools

Command	Key binding	No. of users	Rank
Search for references to selected element in workspace	Ctrl+Shift+G	33	50
Navigate to a type	Ctrl+Shift+T	28	29
Open a type in the hierarchy view	F4	27	119
Open declaration of selected element	F3	26	21
Navigate to last edit location	Ctrl+Q	20	128
Navigate back among open editors	Alt+Left	19	28
Search for declarations of selected element in workspace	Ctrl+G	17	245
Navigate forward among open editors	Alt+Right	14	51
Search for references in a project	n/a	16	96

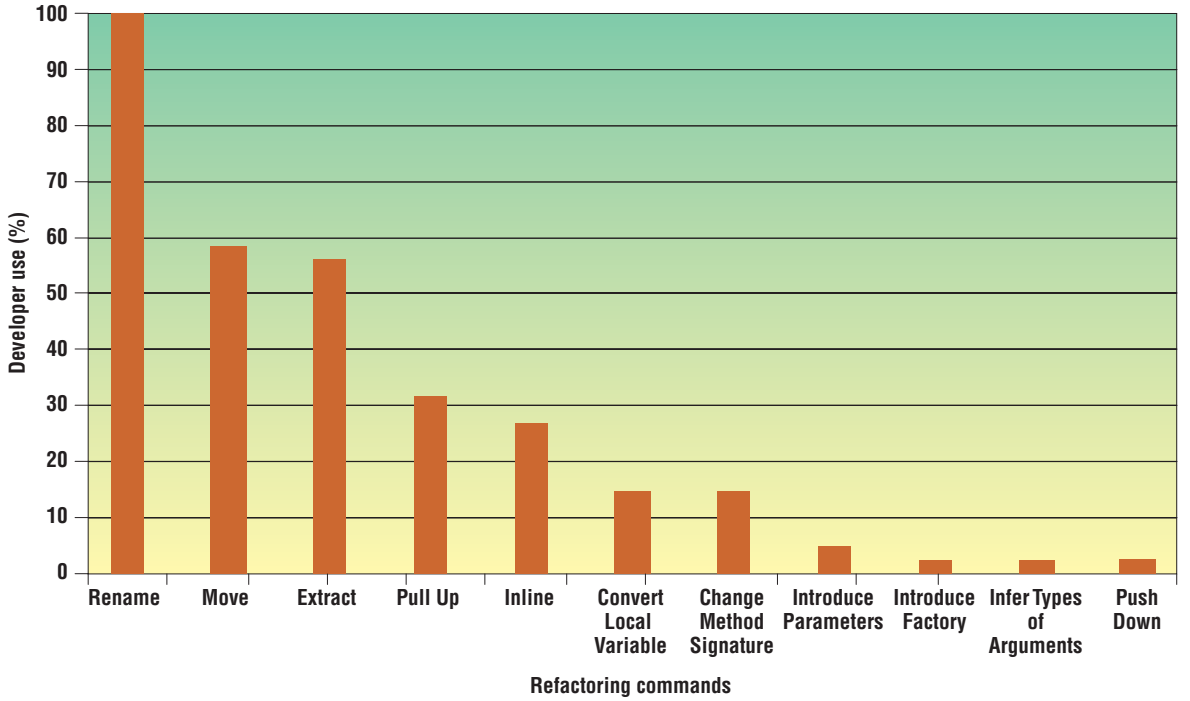


Figure 6. The percentage of developers using 11 categories of refactoring commands.

in files and navigate back to those points, Eclipse provides bookmarking. When viewing a file or a point of interest in a file, a developer can use a pop-up menu to remember the bookmark. A Bookmarks view helps developers access saved bookmarks, but only five developers used it, ranking it 213th.

Refactoring

Development environments are increasingly providing automated support for code refactoring.[3] By examining the commands in the default *Refactor* menus, we found evidence of the developers using 11 kinds of refactoring commands. Figure 6 shows the percentage of users invoking each of these commands.

Developers can invoke each refactoring in different ways, through key bindings, from a context menu in the editor or in a view, or through a pull-down menu. Figure 7 shows how each of the refactoring commands performed by more than 25 percent of the users was invoked. The most common way was

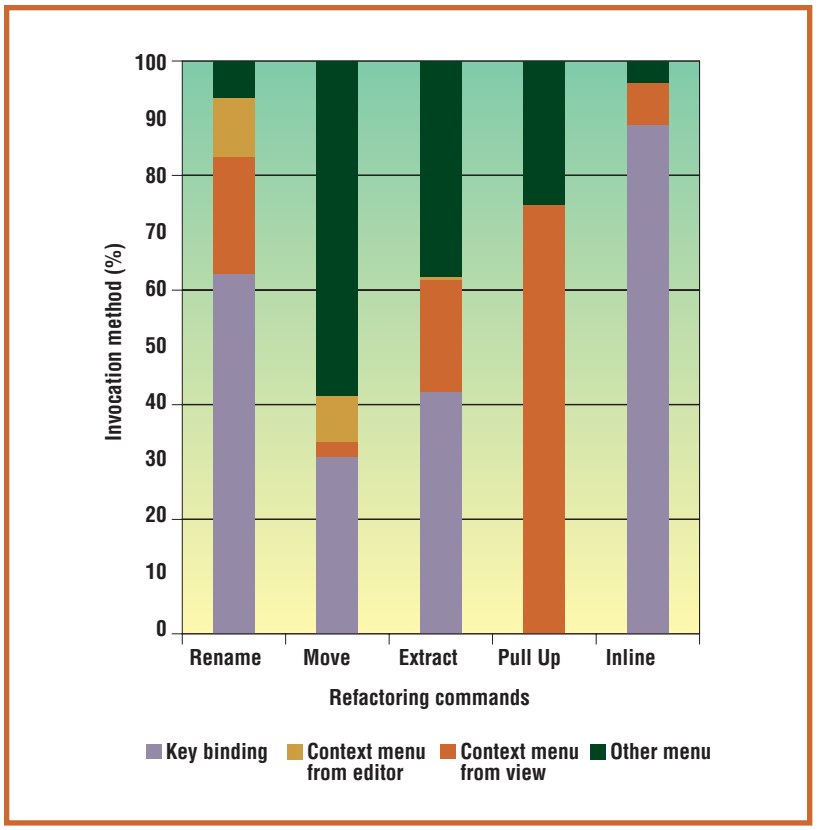


Figure 7. How developers invoked the top five refactoring commands (those used by more than 25 percent of the developers).

through key bindings, which represented 57 percent of the total refactoring invocations. The one exception in figure 7 is Pull Up, which moves methods and fields up the class hierarchy. Because this refactoring command changes the structure of more than one class, it's not surprising developers invoked it more often through menu selections, because it requires more thought about the program's structure.

Debugging views

All but three developers used some of the six debugging views (see figure 8). Almost all of them (over 90 percent) used

- the Variables view, which supports the inspection of variable values, and
- the Debug view, which displays information about the running threads and stacks and provides execution control.

Over 70 percent used the Breakpoints view. Less-used views were the Expressions view, which shows a tree of expressions and their values; the Display view, which helps users evaluate Java code snippets; and, not surprisingly, the Registers view, which helps debug C code.

Source repository use

The developers used three different kinds of source repositories. Most (23 of the 41) used repositories based on CVS (Concurrent Versions System; http://ximbiot.com/cvs/wiki). Almost a quarter of the developers used repositories based on Subversion (http://subversion.tigris.org), and two used a repository based on Microsoft's Visual SourceSafe.

Third-party plug-ins

On the basis of the names given to events in the 12 of 14 interaction histories, it appears that all the developers used a third-party plug-in. On average, 27 percent of the events in an interaction history came from plug-ins that aren't deployed with the default Eclipse download, the Eclipse SDK, which includes the JDT.

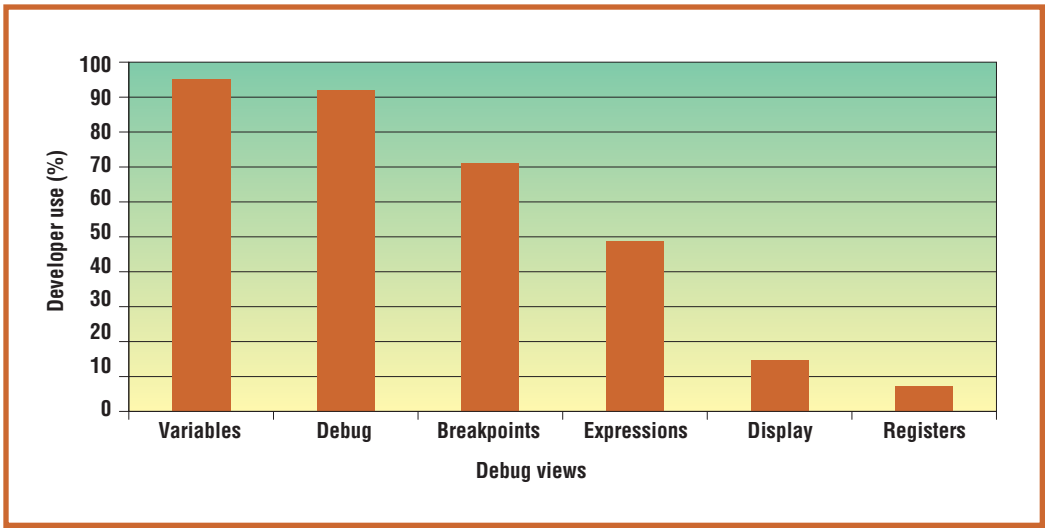


Figure 8. The percentage of the 41 developers using each of the six debugging views.

Despite the long history of software development environments, we haven't been able to find any similar field usage data for one of these environments. Published accounts have focused on the use of a subset of features in the context of laboratory experiments.[4,5] We believe that this sort of feature-usage analysis is becoming increasingly important, as the recent trajectory of these environments has been to provide an ever-increasing amount of functionality.

Our usage monitoring approach allows tool builders to sample how developers are using their tools in the wild. The data gathered about tool use can be used to prevent feature bloat and to evolve the environments according to user needs. Information about how developers work in a development environment can also provide a baseline for assessing new software development tools. We hope this report will provide a start in defining which information to collect and distribute on an ongoing basis to help improve Eclipse and other similar platforms and tools.

Acknowledgments

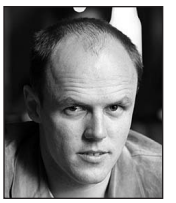
IBM and the Natural Sciences and Engineering Research Council of Canada funded this work. Many thanks to Joanna McGrenere for her helpful comments on an earlier draft.

References

1. C. Zetie. "Eclipse Has Won—What's Next for Eclipse?" *Forrester Research*, 2005; www.forrester.com/Research/Document/Excerpt/0,7211,36165,00.html.
2. G. Goth, "Beware the March of This IDE: Eclipse Is Overshadowing Other Tool Technologies," *IEEE Software*, vol. 22, no. 4, 2005, pp. 108–111.
3. M. Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
4. M.P. Robillard, W. Coelho, and G.C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Trans. Software Eng.*, vol. 30, no. 12, 2004, pp. 889–903.
5. A.J. Ko, H. Aung, and B.A. Myers, "Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), ACM Press, 2005, pp. 126–135.

About the Authors

Gail C. Murphy is an associate professor in the University of British Columbia's Department of Computer Science. Her research interests include software evolution and information structure. She received her PhD in computer science and engineering from the University of Washington. She's a member of the ACM and the IEEE Computer Society. Contact her at the Dept. of Computer Science, Univ. of British Columbia, 201-2366 Main Mall, Vancouver BC Canada; murphy@cs.ubc.ca.



Mik Kersten is a PhD student in the University of British Columbia's Department of Computer Science, lead of the Mylar eclipse.org project, and committer on the AspectJ and AspectJ Development Tools projects. While working at Xerox Palo Alto Research Center, he created integrated development environment support for AspectJ. He now focuses on helping Eclipse reduce information overload by making explicit the context of the tasks we work on. He's a student member of the ACM. Contact him at the Dept. of Computer Science, Univ. of British Columbia, 201-2366 Main Mall, Vancouver BC Canada; beatmik@acm.org.

Leah Findlater is a PhD student in the University of British Columbia's Department of Computer Science. Her research interest is human-computer interaction, focusing on the personalization of complex user interfaces. Contact her at the Dept. of Computer Science, Univ. of British Columbia, 201-2366 Main Mall, Vancouver BC Canada; lkf@cs.ubc.ca.