# A Practical Guide to Analyzing IDE Usage Data

Will Snipes, Emerson Murphy-Hill, Thomas Fritz, Mohsen Vakilian
Kostadin Damevski, Anil R. Nair, David Shepherd

**Abstract**

Integrated Development Environments (IDEs) such as Eclipse and Visual Studio provide tools and capabilities to perform tasks such as navigating among classes and methods, continuous compilation, code refactoring, automated testing, and integrated debugging all designed to increase productivity. Instrumenting the IDE to collect usage data provides a more fine-grained understanding on developers' work than previously possible. Usage data supports analysis of how developers spend their time, what activities might benefit from greater tool support, where developers have difficulty comprehending code, and whether they are following specific practices such as test-driven development. With usage data, we expect to uncover more nuggets of how developers create mental models, how they investigate code, how they perform mini trial-and-error experiments, and what might drive productivity improvements for everyone.

## 1. Introduction

As software development evolved, many developers began using Integrated Development Environments (IDEs) to help manage the complexity of software programs. Modern IDEs such as Eclipse and Visual Studio include tools and capabilities to improve developer productivity by assisting with tasks such as navigating among classes and methods, continuous compilation, code refactoring, automated testing, and integrated debugging. The breadth of development activities supported by the IDE makes collecting editor, command, and tool usage data valuable for analyzing developers' work patterns.

Instrumenting the IDE involves extending the IDE within a provided API framework. Eclipse and Visual Studio support a rich API framework allowing logging of many commands and actions as they occur. We discuss tools that leveraging this API to observe all the commands developers use, developer actions within the editor such as browsing or inserting new code, and other add-in tools developers use. In Section 3, we provide a how-to guide for implementing tools that collect usage data from Eclipse or Visual Studio.

Collecting IDE usage data provides an additional view of how developers produce software to help advance the practice of software engineering. The most obvious application of usage data is to analyze how developers spend their time in the IDE by classifying the events in the usage log and tracking time between each event. Through usage data analysis, we gain a better understanding of the developer's time allocation and can identify opportunities to save time such as reducing build time or improving source code search and navigation tools. Beyond time analysis, researchers have applied usage data to quantify developers use of practices such as the study of types of refactoring by Murphy-Hill, Parnin and Black (2012*a*) that found developers mostly perform minor refactoring while making other changes. In another example, Carter and Dewan (2010) leverage usage data to discover areas of the code where developers have difficulty with comprehension and should ask for assistance from more experienced developers. One study determined whether developers are doing test driven development properly by writing the tests first then writing code to make those tests pass or are doing it improperly by writing tests against previously written code (Kou et al., 2010). Maalej et al. (2014) describe how to collect and process data for recommendation systems including tools and analysis methods, and they discuss important findings from developer usage data analysis. These works provide good examples of how usage data provides necessary information to answer interesting research questions in software engineering.

There are limits, however, to what IDE usage data can tell us. The missing elements include the developer's mental model of the code, and how they intend to alter the code to suit new requirements. We must also separately obtain data on the developers' experience, design ideas, and constraints they keep in mind during an implementation activity.

Looking forward, usage data from development environments provides a platform for greater understanding of low-level developer practices. We expect to uncover more nuggets of how developers work to comprehend source code, how they perform mini trial and error experiments, and what might release further productivity improvements for all developers.

## 2. Usage Data Research Concepts

In this section we discuss background on usage data research and provide motivation for analyzing usage data by describing what we can learn from it. With a review of Goal-Question-Metric, we discuss how to focus usage data collection with specific research goals. To round out the concepts, we discuss additional considerations such as privacy and additional data sources that may useful.

### *2.1. What Is Usage Data and Why Analyze It?*

We refer to the data about the interactions of software developers with an IDE as the *IDE usage data* or simply *usage data*. The interactions include commands invoked, files viewed, mouse clicks, and add-on tools used.

Several stakeholders benefit from capturing and analyzing usage data. First, IDE vendors leverage the data to get insight into ways to improve their product based on how developers use the IDE in practice. Second, researchers both develop usage data collectors and conduct rigorous experiments to (1) make broader contributions to our understanding of developers' coding practices and (2) improve the state-of-the-art programming tools (e.g.,debuggers and refactoring tools). Finally, developers benefit from the analysis conducted on the usage data because these analyses lead to more effective IDEs that make developers more productive.

At a high level, an IDE can be modeled as a complex state machine. In this model, a developer performs an action at each step that moves the IDE from one state to another. To capture usage data, researchers and IDE vendors have developed various usage data collectors (Section 3). Depending on the goals of the experiments, the usage data collector captures data about a subset of the IDE's state machine. While a combination of video recordings of the IDE with the keyboard strokes and mouse clicks of a developer would provide a fairly complete set of usage data, it is difficult to automatically analyze video data and therefore mostly limited to small lab studies and not part of the developed usage data collectors.

An example of a usage data collection and analysis project with wide adoption in practice is the Mylyn project (previously known as Mylar). Mylyn started as a research project that later became part of Eclipse and that exhibits both of the advantages of understanding programmer's practices and improving tool support.

Mylyn by Kersten and Murphy (2005) was one of the first usage data collectors in IDEs. It was implemented as a plug-in for the Eclipse IDE and captured developers' navigation histories and their command invocations. For example, it records changes in selections, views, perspectives as well as invocations of commands such as delete, copy, and automated refactoring. By now, the Mylyn project ships with the official distribution of Eclipse.

Studies, e.g. (Murphy et al., 2006), used the Mylyn project to collect and then analyze usage data to gather empirical evidence on the usage frequency of various features of Eclipse. In addition to collecting usage data, Mylyn introduces new features to the Eclipse IDE that leverages the usage data to provide a task-focused User Interface (UI) and increase a developer's productivity (Kersten and Murphy, 2006). In particular, Mylyn introduces the concept of a *task context*. A task context comprises a developer's interactions in the IDE that are related to the task, such as selections and edits of code entities (e.g.,, files, classes, and packages). Mylyn analyzes the interactions for a task and uses the information to surface relevant information with less clutter in various features such as outline, navigation, and auto-completion. More information on collecting data from Mylyn is in Section 3.2.

Later, Eclipse incorporated a system similar to Mylyn, called the Eclipse Usage Data Collector (UDC)[1], as part of the Eclipse standard distribution package for several years. UDC collected data from hundreds of thousands of Eclipse users every month. To the best of our knowledge, the UDC data set[2] is the largest set of IDE usage data that is publicly available. As described in (Murphy-Hill, Jiresal and Murphy, 2012) and (Vakilian et al., 2013), several papers including (Vakilian and Johnson, 2014;

[1] `http://www.eclipse.org/epp/usagedata/`
[2] `http://archive.eclipse.org/projects/usagedata/`

Vakilian et al., 2013; Murphy-Hill, Parnin and Black, 2012*a*), mined this large data set to gain insight about programmers' practices and develop new tools that better fit programmers' practices. For more information on UDC, see the included Section 3.1 on using UDC to collect usage data from Eclipse.

Studies of automated refactoring are another example of interesting research results from analyzing usage data. Vakilian et al. (2013), and Murphy-Hill, Parnin and Black (2012*a*) analyzed the Eclipse UDC data, developed custom usage data collectors (Vakilian et al., 2012), and conducted survey and field studies, e.g. (Murphy-Hill, Parnin and Black, 2012*a*; Vakilian and Johnson, 2014; Negara et al., 2013), to gain more insight about programmers' use of the existing automated refactorings. Murphy-Hill, Parnin and Black (2012*a*) and Negara et al. (2013) found that programmers do not use the automated refactorings as much as refactoring experts expect. This finding motivated researchers to study the factors that lead to low adoption of automated refactorings in (Vakilian et al., 2012; Murphy-Hill, Parnin and Black, 2012*a*) and propose novel techniques for improving the usability of automated refactorings e.g. (Murphy-Hill, Parnin and Black, 2012*a*; Murphy-Hill, Jiresal and Murphy, 2012; Murphy-Hill and Black, 2008; Lee et al., 2013; Murphy-Hill et al., 2011; Ge et al., 2012; Foster et al., 2012).

With this background on usage data collection and research based on usage data, we look next at how to define usage data collection requirements based on your research goals.

### *2.2. Selecting Relevant Data Based on a Goal*

Tailoring usage data collection to specific needs helps optimize the volume of data and privacy concerns when collecting information from software development applications. While the general solutions described in the next sections collect all events from the Integrated Development Environment (IDE), limiting the data collection to specific areas can make data collection faster and more efficient and reduce noise in the collected data. A process for defining the desired data can follow structures such as Goal-Question-Metric defined by Basili and Rombach (1988) that refines a high-level goal into specific metrics to generate from data. For example, in the Experiences Gamifying Software Development (Snipes et al., 2014) study we focused on the navigation practices of developers. The study tried to encourage developers to use structured navigation practices (navigating source code by using commands and tools that follow dependency links and code structure models). In that study, we defined a subset of the available data based on a Goal-Question-Metric structure as follows:

- Goal

  Assess and compare the use of structured navigation by developers in our study.

- Possible Question(s)

  What is the frequency of navigation commands developers use when modifying source code?

  What portion of navigation commands developers use are structured navigation rather than unstructured navigation?

- Metric

  Navigation Ratio is the proportion of the number of structured navigation commands to the number of unstructured navigation commands used by a developer in a given time period (e.g., a day).

The specific way to measure navigation ratio from usage data needs further refinement to determine how the usage monitor can identify these actions from available events in the IDE. Assessing commands within a time duration (e.g., day) requires, for instance, that we collect a time-stamp for each command. Simply using the time-stamp to stratify the data according to time is then a straight-forward conversion from the time-stamp to a data and grouping the events by day. Similarly the time-stamp can be converted to the hour to look at events grouped by hour of any given day. Calculating duration or elapsed time for a command or set of commands adds new requirements to monitoring. Specifically, the need to collect events like window visibility events from the operating system that relate to when the application or IDE is being used and when it is in the background or closed.

### *2.3. Privacy Concerns*

Usage data can be useful, however, there are some privacy concerns your developers might and often have regarding the data collection and who the data is shared with. These privacy concerns arise mainly since the collected data may expose individual developers or it may expose parts of the source code companies are working on. How you handle information privacy in data collection affects what you can learn from the data during analysis (see Section 4).

To minimize privacy concerns about the collected data, steps such as encrypting sensitive pieces of information, for instance, by using a one-way hash-function can be taken. Hashing sensitive names, such as developer names, window titles, filenames or source code identifiers, provides a way to obfuscate the data and reduce the risk of information that allows identification of the developer or the projects and code they are working on. While this obfuscation makes it more difficult to analyze the exact practices, using a one-way hash-function will still allow differentiation between distinct developers, even if anonymous.

Maintaining developer privacy is important but, there may be questions for which you need the ground truth that confirms what you observe in the usage data. Thus you may need to know who is contributing data so you can ask them questions that establish the ground truth. A privacy policy statement helps participants and developers be more confident sharing information with you when they know they can be identified with the information. The policy statement should specifically state who will have access to the data and what they will do with it. Limiting statements such as not reporting data at the individual level helps to reduce a developer's privacy concerns.

### *2.4. Study Scope*

Small studies that examine a variety of data can generate metrics that you can apply to data collected in bigger studies where the additional information might not be available. For instance, Robillard et al. (2004) defined a metric on structured navigation in their observational study on how developers discover relevant code elements during maintenance. This metric can now be used in a larger industrial study setting in which structured navigation command usage is collected as usage data, even without the additional information Robillard et al. gathered for their study.

Finally and most importantly, usage data may not be enough to definitively solve a problem or inquiry. While usage data tells what a developer is doing in the IDE, it usually leaves gaps in the story (see Section 5). Augmenting usage data with additional data sources such as developer feedback, task descriptions, and change histories (see Section 4.7) can fill in the details necessary to understand user behavior.

Now that we have discussed aspects to consider, we are ready to dig deeper into specifics on how to collect data from developer IDEs. The next Section covers several options for tooling that collects usage data from IDEs.

## 3. How to Collect Data

There are many options for collecting usage data from IDEs. Existing tools can provide solutions for commonly used IDEs and some support collecting data from additional sources. Another way to start is to study data collected in previous projects such as data available in the Eclipse archive for UDC data. This archive contains a wealth of data collected by UDC when UDC was integrated with each Eclipse version in 2009 and 2010. The data is currently available at this URL: `http://archive.eclipse.org/projects/usagedata/`

You may have more specific questions than what can be answered with the UDC data or need to collect usage data for a specific experiment. In this section we discuss details on how to use existing data collection tools for Eclipse including Usage Data Collector, Mylyn Monitor, and CodingSpectator (Section 3.3). Then we'll walk through creating a usage data collection extension to Microsoft Visual Studio. Before we get into details, here is an overview of some existing frameworks.

> Eclipse: Usage Data Collector (UDC) discussed in Section 3.1 collects commands executed in the environment and editors and views that are invoked.

| Tool Name | Advantages | Disadvantages | Examples |
|---|---|---|---|
| Eclipse Usage Data Collector (UDC) | Well tested, widely deployed. | Collects only data on tools; sometimes missing data. | (Liu et al., 2012; Parnin and Rugaber, 2011; Murphy-Hill, Parnin and Black, 2012*b*) |
| Mylyn Monitor | Collects data both about tools and the program elements the tools are used on. | No details about code beyond element names collected. | (Kersten and Murphy, 2006; Ying and Robillard, 2011; Murphy et al., 2009) |
| CodingSpectator | Very detailed information collected. | Information collected largely customized to observe usage of refactoring tools. | (Vakilian et al., 2011, 2012; Negara et al., 2012) |
| Build-Your-Own for Visual Studio | A high degree of customizability. One of the few Visual Studio monitoring tools. | Extra work required to collect a wider variety of events. | (Snipes et al., 2014) |

Table 1: A summary of the four tools discussed in depth in this section.

Eclipse: Mylyn Monitor described in Section 3.2 collects task-oriented events and information about what code elements the programmers work with.

Eclipse: CodingSpectator, discussed in Section 3.3, focuses on refactoring actions and the context in which they are taken.

VisualStudio: The Build It Yourself for Visual Studio Section 3.4 of this chapter describes in detail how to build your own Visual Studio extension that collects all command events from the IDE.

VisualStudio: CodeAlike[3] is a Visual Studio extension for personal analytics and research of usage data related to coding efficiency.

Eclipse: CodingTracker, by Negara et al. (2012), is a usage data collector for the Eclipse IDE that records every character insertion and deletion. CodingTracker records the code edits so accurately that it can later replay them to show the changes in action. CodingTracker has been used to conduct empirical studies and accurately infer high-level changes such as refactorings (Negara et al., 2013).

Eclipse: Fluorite, by Yoon and Myers (2011), is an Eclipse-based tool that captures usage data such as invoked commands, typed characters, cursor movements, and text selections. Fluorite has been used to study programmers' backtracking strategies in (Yoon and Myers, 2012) and visualizing code histories in (Yoon et al., 2013).

Eclipse and Visual Studio: Hackystat by Johnson et al. (2003), provides a framework to collect usage data from many sources.

The remainder of this section will discuss in detail how to implement the top four tools from the list above. Where the section describes code, listings are provided based on the open source code available on GitHub (`https://github.com/wbsnipes/AnalyzingUsageDataExamples`). In Table 1, we summarize the advantages and disadvantages of each of the four tools we discuss in this chapter, as well as example papers that used these tools.

[3] `https://codealike.com`

### *3.1. Eclipse Usage Data Collector*

This section outlines how to collect IDE usage data using Eclipse's Usage Data Collector (UDC).[4] The UDC framework was originally build by the Eclipse Foundation, as a way to measure how the community was using the Eclipse IDE. While UDC was included in official Eclipse releases and data was collected from hundreds of thousands of Eclipse users between 2008 and 2011, the project was eventually shut down, and UDC was removed from official Eclipse releases. However, the source code for UDC remains available for collecting data.

#### *3.1.1. Collected Data*

The Eclipse Data Collector records the following types of Eclipse information:

- The run-time environment, such as the operating system and Java Virtual Machine.
- Environment data, such as which bundles are loaded and when Eclipse starts up and shuts down
- Actions and commands that are executed, via menus, buttons, toolbars, and hotkeys.
- Views, editors, and perspectives that are invoked.

Let's look at an example of an event that UDC produces on a developer's machine:

| **what** | **kind** | **bundleId** | **bundleVersion** | **description** | **time** |
|---|---|---|---|---|---|
| executed | command | org.eclipse.ui | 3.7.0.v20110928-1505 | org.eclipse.ui.edit.paste | 1389111843130 |

The first column tells us what kind of thing happened—in this case, something was executed. The second column tells us what was executed—in this case, a command. The third column tells us what the name of a bundle (a set of resources and code installed in Eclipse) this event belonged to—in this case, Eclipse's user interface bundle. The fourth column gives us the version of the bundle. The fifth tells us the name of the command that was executed—in this case, paste. The final column, a unix time-stamp that tells us when the command was executed, in Greenwich Mean Time—in this case, January 7th, 2014 at 16:24:03 GMT.

#### *3.1.2. Limitations*

Apart from general limitations of collecting usage data (Section 5), one significant limitation of UDC that we have found is that sometimes it has unexpectedly incomplete data. For example, in planning for a study involving when people ran their JUnit tests, we found that UDC recorded an event when the "Run > Run As > Run as JUnit Test" menu item was selected, but not when the "Run As" button was pressed on the toolbar. We suspect that the reason has to do with how different user interface accordances invoke the same functionality. In general, when you are planning on running a study with UDC, be sure to know what types of events you are looking for, and test them to make sure UDC captures those events.

#### *3.1.3. How to Use It*

Collecting usage data is fairly straightforward with Eclipse UDC, and we describe how to do so here. We also include an accompanying screencast that shows the basics.[5]

*Gathering Data Using the UDC Client..* Let's talk about how data is collected on a developer's machine. Since UDC was last included in the Eclipse Indigo SR2 release,[6] if you have the option of which Eclipse to use, we recommend downloading that version. By default, UDC starts collecting data when Eclipse is start up. You can verify this by going to "Windows > Preferences", then select the "Usage Data Collector" item (Figure 1). The *Enable capture* option should be checked.

[4] `http://www.eclipse.org/epp/usagedata/`
[5] `http://youtu.be/du4JTc9UB-g`
[6] (`http://www.eclipse.org/downloads/packages/release/indigo/sr2`

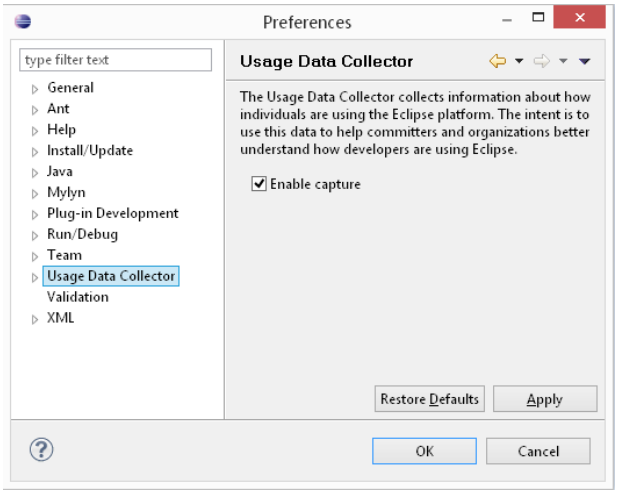


Figure 1: Eclipse Usage Data Collector preference page.

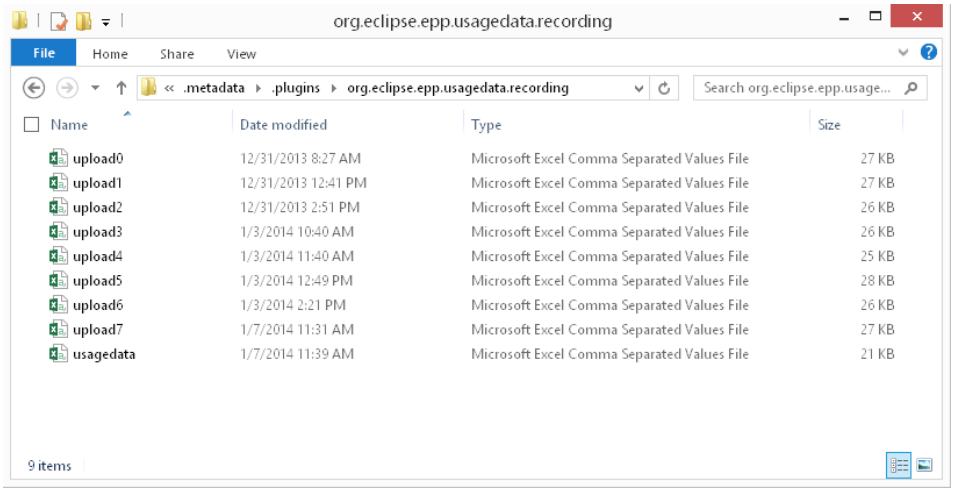


Figure 2: UDC data files.

Before looking at the data, execute a few commands and open a few views in Eclipse. Then, on your file system, open the following path as a subdirectory of your current workspace (Figure 2):

```
1 .metadata/.plugins/org.eclipse.epp.usagedata.recording
```

In that folder, depending on how many UDC events have been gathered, a number of comma separated value (CSV) files will appear, where `upload0.csv` is the oldest and `usagedata.csv` is the newest. Open up `usagedata.csv`—you should notice a large number and a variety of events. Be sure to look specifically for events that you executed and views that you opened earlier.

Before doing a study, be aware that Eclipse will ask and periodically attempt to upload data to the Eclipse foundation server. You should *not* allow it to do this, because each time data is uploaded, the underlying CSV files are deleted. Furthermore, because the UDC project is no longer officially supported, the official Eclipse UDC server no longer accepts the data, so your usage data is, in effect, lost permanently. Unfortunately, there is no easy way to tell the UDC client to permanently store usage data. An easy workaround is to increase the upload period to allow enough time to complete the experiment(see Figure 3). The long-term fix for this issue is to either use some other tool, such as

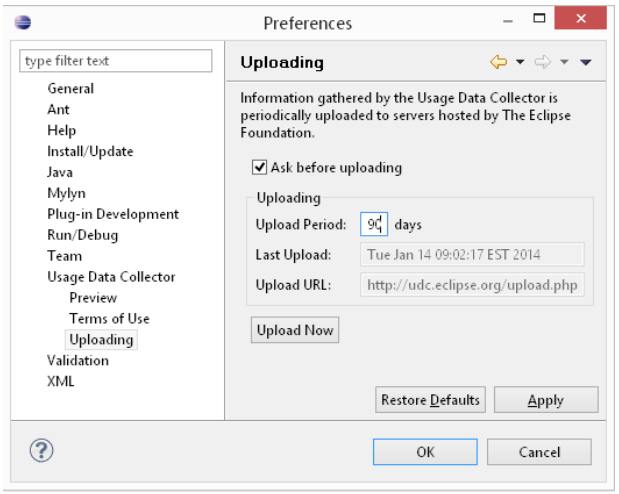


Figure 3: Changing the UDC upload frequency.

CodingSpectator (Section 3.3), to periodically submit the UDC data to your servers or modify the source code of UDC, as we will explain how to do shortly, to never upload data.

If you're doing a laboratory experiment, collecting data should be simply a matter of copying and deleting the CSV files after each participant has done the experiment. You can append the files together or put them in a database for analysis.

*Modifying the UDC Client..* You may wish to modify the UDC client yourself, perhaps to add a custom filter for events or to disable data uploading. Whatever the reason, making modifications to the client is fairly easy.

The first step is to check out the UDC source code into your Eclipse workspace using git.[7] Here we will again use Eclipse Indigo SR2, but we will specifically be using the "Eclipse for RCP and RAP Developers" download package because we will modify Eclipse plugins. Before importing the necessary plugins, we recommend switching to the Indigo SR2 tag, to assure compatibility with Eclipse. To do so, clone the git repository[8] locally, open up "Tags", right click on "Indigo SR 2", then choose "Checkout".

To import the projects into Eclipse, right click on the repository, then click "Import Projects," then "Import Existing Projects." The three core projects to import are:

```
1 org.eclipse.epp.usagedata.internal.gathering
2 org.eclipse.epp.usagedata.internal.recording
3 org.eclipse.epp.usagedata.internal.ui
```

Next, we recommend a quick smoke test to determine whether you can actually make changes to the UDC client. Open `UsageDataRecordingSettings.java`, then modify the value of `UPLOAD_URL_DEFAULT` to `"my_changed_server"`. Then, create a new debug configuration that is an Eclipse Application, and press "Debug" (Figure 4). Finally, you can verify that your change worked by going to UDC's Uploading preference page, noticing that the Upload URL is now "my_changed_server".

From here, you can make any changes to the UDC client that you wish. One thing you may want to do is upgrade UDC to work with more recent versions of Eclipse. The code is likely currently out of date because it has not been maintained since the UDC project was shut down. Another thing you may wish to do is deploy your new version of UDC via an Eclipse update site to the developers you want to study. There are many resources on the web for plugin deployment instructions, such as Lars Vogel's

[7] `http://git-scm.com/`
[8] `http://git.eclipse.org/c/epp/org.eclipse.epp.usagedata.git/`

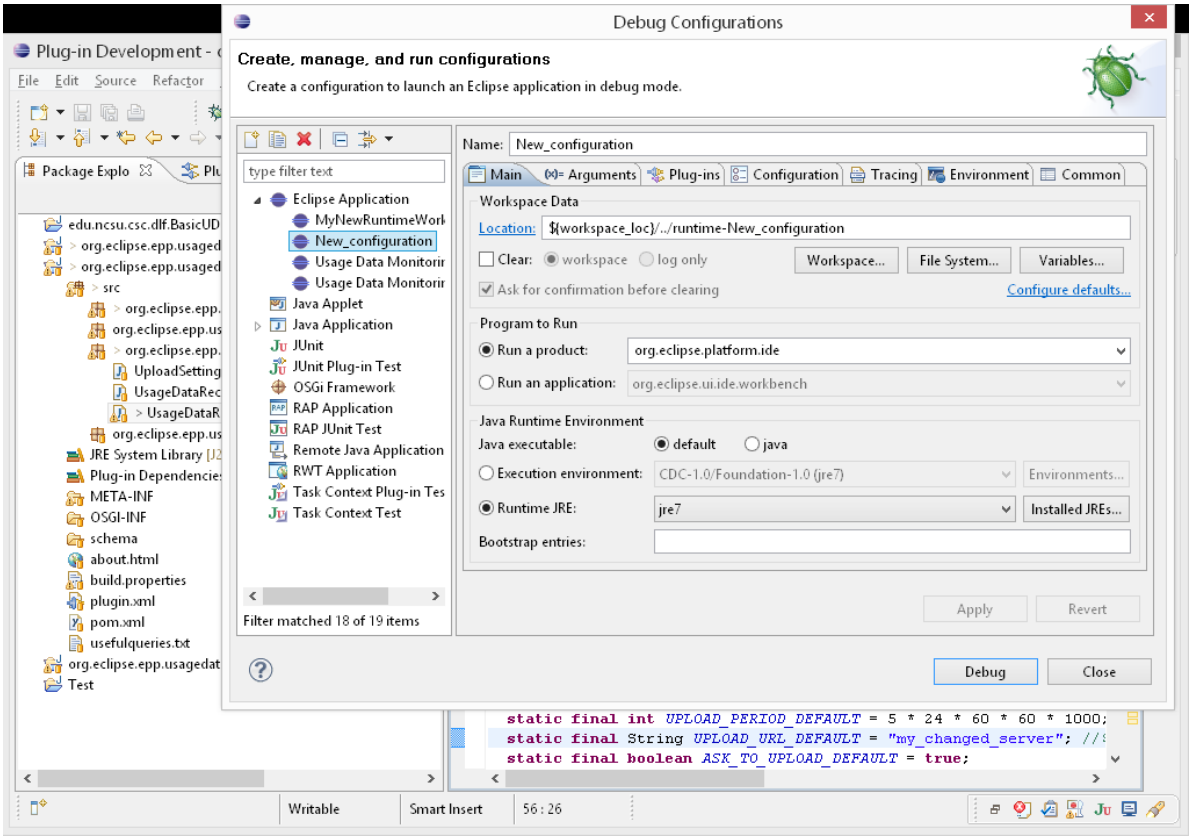


Figure 4: Debugging the UDC client.

tutorial on creating plugins.[9]

*Transmitting Data over the Internet..* If you do not plan on doing a lab study where you can manually collect UDC usage files, you will want to have the UDC client send the data to you directly. As we mentioned, the best way to do this is probably by changing the default server URL in the client source code. An easy way to change the server when debugging is by adding the following Java virtual machine arguments:

```
-Dorg.eclipse.epp.usagedata.recording.upload-url=http://localhost:8080
```

However, simply changing the client to point at a new URL is insufficient, because there actually has to be a working server at that URL, ready to receive UDC data. While the source code of official Eclipse server was not officially made available, Wayne Beaton from the Eclipse Foundation unofficially released some of the PHP code from the Eclipse Foundation's server.[10] Since our PHP skills are rusty, next we'll discuss how to create our own server using Java.

Creating your own server that receives UDC data is fairly straightforward. Let's create a simple one using Apache's HttpComponents library, the same library that UDC uses to upload data. Specifically, we can create a server by simply extending Apache's tutorial web server.[11] You can find this server in our github repository.[12]

First, we'll need a generic request handler to wait for HTTP connections:

```
import java.io.IOException;
import org.apache.http.ConnectionClosedException;
import org.apache.http.HttpException;
import org.apache.http.HttpServerConnection;
import org.apache.http.protocol.BasicHttpContext;
import org.apache.http.protocol.HttpContext;
import org.apache.http.protocol.HttpService;

/**
 * Based on
 * http://hc.apache.org/httpcomponents-core-ga/httpcore/examples/org/apache
 * /http/examples/ElementalHttpServer.java
 */
class WorkerThread extends Thread {

    private final HttpService httpservice;
    private final HttpServerConnection conn;

    public WorkerThread(final HttpService httpservice, final HttpServerConnection conn) {
        super();
        this.httpservice = httpservice;
        this.conn = conn;
    }

    @Override
    public void run() {
        System.out.println("New connection thread");
        HttpContext context = new BasicHttpContext(null);
        try {
            while (!Thread.interrupted() && this.conn.isOpen()) {
                this.httpservice.handleRequest(this.conn, context);
            }
        } catch (ConnectionClosedException ex) {
```

[9] http://www.vogella.com/tutorials/EclipsePlugIn/article.html#deployplugin_tutorial

[10] https://bugs.eclipse.org/bugs/show_bug.cgi?id=221104

[11] http://hc.apache.org/httpcomponents-core-ga/httpcore/examples/org/apache/http/examples/ElementalHttpServer.java

[12] https://github.com/wbsnipes/AnalyzingUsageDataExamples

```
34              System.err.println("Client closed connection");
35          } catch (IOException ex) {
36              System.err.println("I/O error: " + ex.getMessage());
37          } catch (HttpException ex) {
38              System.err.println("Unrecoverable HTTP protocol violation: " + ex.getMessage());
39          } finally {
40              try {
41                  this.conn.shutdown();
42              } catch (IOException ignore) {
43              }
44          }
45      }
46  }
```

We'll also need a generic request listener:

```
import java.io.IOException;
import java.io.InterruptedIOException;
import java.net.ServerSocket;
import java.net.Socket;

import org.apache.http.HttpConnectionFactory;
import org.apache.http.HttpServerConnection;
import org.apache.http.impl.DefaultBHttpServerConnection;
import org.apache.http.impl.DefaultBHttpServerConnectionFactory;
import org.apache.http.protocol.HttpService;

/**
 * Based on
 * http://hc.apache.org/httpcomponents-core-ga/httpcore/examples/org/apache
 * /http/examples/ElementalHttpServer.java
 */
class RequestListenerThread extends Thread {

    private final HttpConnectionFactory<DefaultBHttpServerConnection> connFactory;
    private final ServerSocket serversocket;
    private final HttpService httpService;

    public RequestListenerThread(final int port, final HttpService httpService)
            throws IOException {
        this.connFactory = DefaultBHttpServerConnectionFactory.INSTANCE;
        this.serversocket = new ServerSocket(port);
        this.httpService = httpService;
    }

    @Override
    public void run() {
        System.out.println("Listening on port " + this.serversocket.getLocalPort());
        while (!Thread.interrupted()) {
            try {
                // Set up HTTP connection
                Socket socket = this.serversocket.accept();
                System.out.println("Incoming connection from" + socket.getInetAddress());
                HttpServerConnection conn = this.connFactory.createConnection(socket);

                // Start worker thread
                Thread t = new WorkerThread(this.httpService, conn);
                t.setDaemon(true);
                t.start();
            } catch (InterruptedIOException ex) {
                break;
            } catch (IOException e) {
                System.err.println("I/O error initialising connection thread: " + e.getMessage());
                break;
            }
        }
    }
}
```

And finally, the guts of our server:

```
import java.io.IOException;
import org.apache.http.HttpEntityEnclosingRequest;
import org.apache.http.HttpException;
import org.apache.http.HttpRequest;
import org.apache.http.HttpResponse;
import org.apache.http.protocol.HttpContext;
import org.apache.http.protocol.HttpProcessor;
import org.apache.http.protocol.HttpProcessorBuilder;
import org.apache.http.protocol.HttpRequestHandler;
import org.apache.http.protocol.HttpService;
import org.apache.http.protocol.ResponseConnControl;
import org.apache.http.protocol.ResponseContent;
import org.apache.http.protocol.ResponseDate;
import org.apache.http.protocol.ResponseServer;
import org.apache.http.protocol.UriHttpRequestHandlerMapper;
import org.apache.http.util.EntityUtils;

/**
 * Based on
 * http://hc.apache.org/httpcomponents-core-ga/httpcore/examples/org/apache
 * /http/examples/ElementalHttpServer.java
 */
public class BasicUDCServer {

	public static void main(String[] args) throws IOException {

	int port = 8080;

	HttpProcessor httpproc = HttpProcessorBuilder.create()
			.add(new ResponseDate()).add(new ResponseServer())
			.add(new ResponseContent()).add(new ResponseConnControl()).build();

	UriHttpRequestHandlerMapper reqistry = new UriHttpRequestHandlerMapper();
	reqistry.register("*", new HttpRequestHandler() {

	    public void handle(HttpRequest request, HttpResponse response,
	            HttpContext context) throws HttpException, IOException {

	        HttpEntityEnclosingRequest entityRequest = (HttpEntityEnclosingRequest) request;

	        String userID = request.getHeaders("USERID")[0].getValue();
	        String workspaceID = request.getHeaders("WORKSPACEID")[0].getValue();
	        long time = Long.parseLong(request.getHeaders("TIME")[0].getValue());

	        System.out.println(userID + "," + workspaceID + "," + time);
	        System.out.println(EntityUtils.toString(entityRequest.getEntity()));
	    }
	});

	HttpService httpService = new HttpService(httpproc, reqistry);

	Thread t = new RequestListenerThread(port, httpService);
	t.setDaemon(false);
	t.start();
    }
}
```

When this server is running and it receives a UDC upload, it will print a UserId, WorkspaceId, and the time of the upload. UserIds are randomly generated on the client side and stored in a file in the developer's home directory. As long as that file remains intact, future uploads from that developer will contain that UserId. WorkspaceIds are identifiers contained in each workspace, and can be used to uniquely (but anonymously) identify which workspace a set of data is uploaded from. Thus, there is

normally only one UserId per computer, but there can be multiple WorkspaceIds per computer.

While there is some coding involved, setting up the UDC for Eclipse can provide thorough usage data collection for a project using Eclipse. For lab studies, not much setup is required. For larger or distributed studies, some infrastructure (a web server) and code (the UDC data server) are required. Next we look at Mylyn and the Eclipse Mylyn Monitor component, which collects tool data like UDC, but also includes information about what program elements the programmer is working with.

### *3.2. Mylyn and the Eclipse Mylyn Monitor*

Kersten and Murphy (2006) created Mylyn, a task focused user interface, a top-level project of the Eclipse IDE that is part of many of the Eclipse IDE configurations. To better support developers in managing and working on multiple tasks, Mylyn makes tasks a first class entity, monitors a developer's interaction with the IDE for each task and logs it in a so-called *task context*.

The first versions of Mylyn, originally called Mylar, were developed as part of the PhD research of Mik Kersten and contained an explicit Mylyn Monitor component to collect and upload a developer's activity within the Eclipse IDE. While the source code of the Mylyn Monitor can still be found on-line, it is not an active part of the Mylyn project anymore.

#### *3.2.1. Collected Data*

Mylyn captures three types of developer interactions with the Eclipse development environment:

- the *selection* of elements,
- the *editing* of elements, and
- *commands* in the IDE, such as saving or refactoring commands.

These interaction events are monitored and then stored in XML format in a log file. An interaction event log example of a developer selecting a Java class `TaskEditorBloatMonitor.java` in the package explorer of the Eclipse IDE is:

```
<InteractionEvent
    StructureKind="java"
    StructureHandle="=org.eclipse.mylyn.tasks.ui/src&lt;org.eclipse.mylyn.
        internal.tasks.ui{TaskEditorBloatMonitor.java"
    StartDate="2012-04-10 02:05:53.451 CEST"
    OriginId="org.eclipse.jdt.ui.PackageExplorer"
    Navigation="null"
    Kind="selection"
    Interest="1.0"
    EndDate="2012-04-10 02:05:53.451 CEST"
    Delta="null"
/>
```

The log entry contains among other information the interaction event kind, in this case a selection, the full identifier of the element the developer interacted with, a Java type called TaskEditorBloatMonitor, the time when the interaction event occurred, in this case October 4th, 2012, at 02:05:52 CEST, and the place where the interaction event occurred, in this case the package explorer view of Eclipse.

You may notice that the log also contains an interest value, in this case 1.0. This value is used by Mylyn to calculate the interest a developer shows in a code element, the so-called degree-of-interest. The degree-of-interest of a code element, such as a class, method or field, is based on the recency and frequency of interactions while working on a task. The more frequent and recent a developer selected and/or edited a code element, the higher the degree-of-interest. This degree-of-interest value is then used to highlight and/or filter elements in views of the IDE (Kersten, 2007; Kersten and Murphy, 2006, see).

*3.2.2. Logging Interactions with the Mylyn Monitor*

While the code for the Mylyn Monitor is not part of the active Mylyn project anymore, the code for the monitor and example code for using it can be found in the incubator project online[13] [14]. In the following, we will present relevant parts of the code from these examples to log the interactions.

To be able to use the Mylyn Monitor code and log the events of interest there are two important classes you have to implement. First, you will need a plug-in class that extends the following plugin:

```
org.eclipse.ui.plugin.AbstractUIPlugin
```

Then, add a listener for the events that you are interested in to:

```
org.eclipse.mylyn.internal.monitor.ui.MonitorUiPlugin
```

Second, you will need to write the listener that creates the interaction event objects when an interaction event occurs. Let's assume you want to write a listener for selections of Java elements in the IDE. In this case you can extend the class `org.eclipse.mylyn.monitor.ui.AbstractUserInteractionMonitor` and simply override the `selectionChanged` method. By extending the `AbstractUserInteractionMonitor`, your listener will automatically be added as a post selection listener to all windows in the current workbench so that all selection events in the windows are forwarded to your listener. The relevant code for the `selectionChanged` method is:

```
/**
 * Based on
 * http://git.eclipse.org/c/mylyn/org.eclipse.mylyn.incubator.git/tree/
 * org.eclipse.mylyn.examples.monitor.study/src/org/eclipse/mylyn/examples/
 * monitor/study/SelectionMonitor.java
 */
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.mylyn.monitor.core.InteractionEvent;
import org.eclipse.jdt.core.IJavaElement;

...

    @Override
    public void selectionChanged(IWorkbenchPart part, ISelection selection) {
      InteractionEvent.Kind interactionKind = InteractionEvent.Kind.SELECTION;
      if (selection instanceof StructuredSelection) {
          StructuredSelection structuredSelection = (StructuredSelection) selection;
          Object selectedObject = structuredSelection.getFirstElement();
          if (selectedObject == null) {
              return;
          }

          if (selectedObject instanceof IJavaElement) {
              IJavaElement javaElement = (IJavaElement) selectedObject;
              structureKind = STRUCTURE_KIND_JAVA;
              elementHandle = javaElement.getHandleIdentifier();
          }
      }

      ...

      InteractionEvent event = new InteractionEvent(interactionKind, structureKind,
                                  elementHandle, ...);
      MonitorUiPlugin.getDefault().notifyInteractionObserved(event);
   }
```

[13] `http://git.eclipse.org/c/mylyn/org.eclipse.mylyn.incubator.git/tree/`
[14] `http://wiki.eclipse.org/Mylyn_Integrator_Reference#Monitor_API`

```
37
38 ...
```

The code first checks what type the selection has. If the selection is structured, and the first part of it is a Java Element, it collects the relevant information and then creates an `InteractionEvent` with the gathered information, such as the interaction kind, the structure kind and the element handle. At the end of the method, the `MonitorUiPlugin` is notified about the observed interaction event. The `MonitorUiPlugin` will then go through all registered interaction event listeners and forward the event to them. Since there is an `InteractionEventLogger` registered as part of the Mylyn code, the interaction event object will be forwarded to the logger and then written out into a file.

### *3.3. CodingSpectator*

CodingSpectator[15] is an extensible framework for collecting Eclipse usage data. Although researchers at the University of Illinois at Urbana-Champaign developed CodingSpectator primarily for collecting detailed data about the use of the Eclipse refactoring tool, it also provides a reusable infrastructure for *submitting usage data* from developers to a central repository. CodingTracker[16] described by (Negara et al., 2012, 2013) is another data collector developed at Illinois, which collects finer-grained IDE actions while reusing the data submission infrastructure provided by CodingSpectator.

#### *3.3.1. Collected Data*

CodingSpectator was designed for capturing detailed data about the use of automated refactorings. It collects three kinds of refactoring events: `canceled`, `performed`, and `unavailable`. If a programmer starts an automated refactoring but quits it before it finishes, CodingSpectator records a `canceled` refactoring event. If a programmer applies an automated refactoring, CodingSpectator records a `performed` refactoring event. Finally, if a programmer invokes an automated refactoring but the IDE refuses to start the automated refactoring indicating that the refactoring is not applicable to the selected program element, CodingSpectator records an `unavailable` refactoring event.

Eclipse creates a *refactoring descriptor* object for each `performed` refactoring events and serializes it in an XML file. CodingSpectator saves more data in Eclipse refactoring descriptors of `performed` refactorings. In addition, it creates and serializes refactoring descriptors for `canceled` and `unavailable` refactoring events. CodingSpectator supports 23 of the 33 automated refactorings that Eclipse supports.

We show a concrete example of the data that CodingSpectator collects for an invocation of the automated Extract Method refactoring in Eclipse, which extracts a piece of code into a new method. This refactoring moves a selected piece of code into a new method and replaces the selected code by an invocation to the new method. To use the automated Extract Method refactoring, a programmer has to go through multiple steps. First, the programmer selects a piece of code (Figure 5). Second, the programmer invokes the automated Extract Method and configures it (Figure 6). In this case, the programmer sets the name of the new method. The configuration page provides a number of other options including method accessibility, the ordering and names of method parameters, and the generation of method comments. Third, after configuring the refactoring, the programmer hits the "Preview" button and the automated refactoring reports the problems that the refactoring may introduce (Figure 7). In this example, the automated refactoring complains that the selected name of the new method conflicts with the name of an existing method. Finally, the programmer decides to cancel the refactoring and CodingSpectator records a refactoring descriptor for this `canceled` refactoring, as shown in Figure 8. The type of a refactoring event (i.e., `unavailable`, `canceled`, and `performed`) can be inferred from the directory in which the XML file containing the refactoring descriptor resides. CodingSpectator captures the following attributes for the canceled automated Extract Method refactoring in the above example.

1. `captured-by-codingspectator`: indicates that CodingSpectator created the refactoring descriptor.

[15] `http://codingspectator.cs.illinois.edu/`
[16] `http://codingtracker.web.engr.illinois.edu/`

```
package org.elasticsearch.action.support.single.custom;

import org.elasticsearch.action.ActionRequestBuilder;

/**
 */
public abstract class SingleCustomOperationRequestBuilder<Request extends SingleCustomOperationRequest<Request>, Response extends ActionResponse
        extends ActionRequestBuilder<Request, Response, RequestBuilder> {

    protected SingleCustomOperationRequestBuilder(InternalGenericClient client, Request request) {
        super(client, request);
    }

    /**
     * Controls if the operation will be executed on a separate thread when executed locally.
     */
    @SuppressWarnings("unchecked")
    public final RequestBuilder setOperationThreaded(boolean threadedOperation) {
        request.operationThreaded(threadedOperation);
        return (RequestBuilder) this;
    }

    /**
     * if this operation hits a node with a local relevant shard, should it be preferred
     * to be executed on, or just do plain round robin. Defaults to <tt>true</tt>
     */
    @SuppressWarnings("unchecked")
    public final RequestBuilder setPreferLocal(boolean preferLocal) {
        request.preferLocal(preferLocal);
        return (RequestBuilder) this;
    }
}
```

Figure 5: A programmer selects a piece of code to extract into a new method. The selected code is part of class `SingleCustomOperationRequestBuilder` from commit `bdb1992` of the open-source Elasticsearch project (`https://github.com/elasticsearch/elasticsearch`).

2. `stamp`: a time-stamp recording when the refactoring event occurred
3. `code-snippet`, `selection`, `selection-in-code-snippet`, `selection-text`: the location and contents of the selection that the programmer made before invoking the automated refactoring
4. `id`: the automated refactoring's identifier
5. `comment`, `description`, `comments`, `destination`, `exceptions`, `flags`, `input`, `name`, `visibility`: configuration options, e.g., input elements, project, and settings that programmers can set to control the effect of the refactoring
6. `status`: any problems reported by the automated refactoring to the programmer
7. `navigation-history`: when the programmer pressed a button to navigate from one page of the refactoring wizard to another
8. `invoked-through-structured-selection`, `invoked-by-quick-assist`: selection method (e.g., structured or textual selection and whether the automated refactoring was invoked using Quick Assist

### *3.3.2. Deploying CodingSpectator*

Deploying CodingSpectator consists of two main steps: (1) setting up a Subversion repository and (2) setting up an Eclipse update site.

1. **Setting Up a Subversion Repository -** CodingSpectator regularly submits developers' data to a central Subversion repository. To collect CodingSpectator's data automatically, you need to set up a Subversion repository and create accounts for your developers. To allow the developers to submit their data to the Subversion repository, you should grant them appropriate write accesses to the repository.

   Using a Version Control System such as Subversion as the data repository has several advantages:

   (a) Subversion makes all revisions of each file easily accessible. This makes troubleshooting easier.

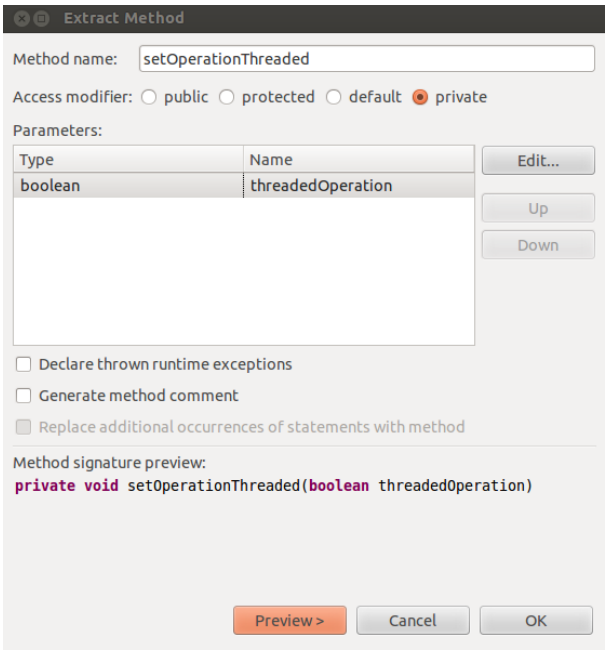


Figure 6: A programmer configures an automated Extract Method refactoring by entering the desired name of the new method.

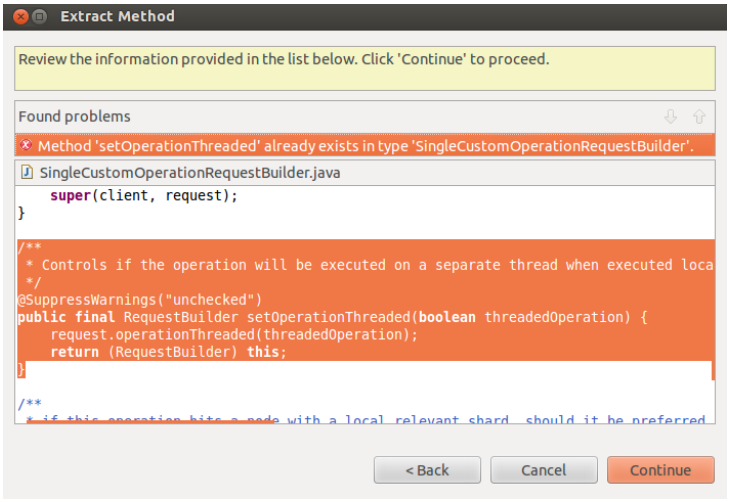


Figure 7: The Extract Method refactoring reports a name conflict problem to the programmer. The programmer can either ignore the problem and continue the refactoring, go back to the configuration page to provide a different name, or cancel the refactoring.

| Node | Content |
| --- | --- |
| xml | version="1.0" encoding="UTF-8" |
| session | |
| version | 1.0 |
| refactoring | |
| captured-by-codingspectator | true |
| code-snippet | /**<br>*/<br>public abstract class SingleCustomOperationRequestBuilder<Request extends SingleCustomOperationReque<br>extends ActionRequestBuilder<Request, Response, RequestBuilder> {<br><br>protected SingleCustomOperationRequestBuilder(InternalGenericClient client, Request request) {<br>super(client, request);<br>}<br><br>/**<br>* Controls if the operation will be executed on a separate thread when executed locally.<br>*/<br>@SuppressWarnings("unchecked")<br>public final RequestBuilder setOperationThreaded(boolean threadedOperation) {<br>request.operationThreaded(threadedOperation);<br>return (RequestBuilder) this;<br>}<br><br>/**<br>* if this operation hits a node with a local relevant shard, should it be preferred<br>* to be executed on, or just do plain round robin. Defaults to <tt>true</tt><br>*/<br>@SuppressWarnings("unchecked")<br>public final RequestBuilder setPreferLocal(boolean preferLocal) {<br>request.preferLocal(preferLocal);<br>return (RequestBuilder) this;<br>}<br>} |
| comment | Extract method 'private void setOperationThreaded(boolean threadedOperation)' from 'org.elasticsearch.ac<br>- Original project: 'elasticsearch'<br>- Method name: 'setOperationThreaded'<br>- Destination type: 'org.elasticsearch.action.support.single.custom.SingleCustomOperationRequestBuilder'<br>- Declared visibility: 'private' |
| comments | false |
| description | Extract method 'setOperationThreaded' |
| destination | 0 |
| exceptions | false |
| flags | 786434 |
| id | org.eclipse.jdt.ui.extract.method |
| input | /src\/main\/java<org.elasticsearch.action.support.single.custom{SingleCustomOperationRequestBuilder.java |
| invoked-by-quickassist | false |
| invoked-through-structured-selection | false |
| name | setOperationThreaded |
| navigation-history | {[Extract Method,BEGIN_REFACTORING,1390255987858],[ExtractMethodInputPage,Previe&w >,139025599846 |
| parameter1 | boolean threadedOperation threadedOperation |
| replace | false |
| selection | 1710 45 |
| selection-in-code-snippet | 697 45 |
| selection-text | request.operationThreaded(threadedOperation); |
| stamp | 1390256000993 |
| status | <ERROR<br><br>ERROR: Method 'setOperationThreaded' already exists in type 'SingleCustomOperationRequestBuilder'.<br>Context: [Working copy] SingleCustomOperationRequestBuilder.java [in org.elasticsearch.action.support.sing<br>package org.elasticsearch.action.support.single.custom<br>import org.elasticsearch.action.ActionRequestBuilder<br>import org.elasticsearch.action.ActionResponse<br>import org.elasticsearch.client.internal.InternalGenericClient<br>class SingleCustomOperationRequestBuilder<br>SingleCustomOperationRequestBuilder(InternalGenericClient, Request)<br>RequestBuilder setOperationThreaded(boolean)<br>RequestBuilder setPreferLocal(boolean)<br>code: none<br>Data: null<br>> |
| version | 1.0 |
| visibility | 2 |

Figure 8: An example refactoring descriptor recorded by CodingSpectator.

(b) For textual files, Subversion submits only the *changes* made to the files as opposed to the entire new file. This differential data submission leads to faster submissions.

(c) There are libraries such as SVNKit[17] that provide an API for Subversion operations such as add, update, remove, and commit. CodingSpectator uses SVNKit for submitting developers' data to the central repository.

(d) Setting up a Subversion server is a well-documented process. This avoids the burden of setting up a specialized server.

On the other hand, a disadvantage of using Subversion as the data repository is that it requires the developers to maintain a copy of their data on their file systems. The Subversion working copy on the developers' systems takes *space* and can also cause *merge conflicts*, e.g.,, if a developer restores the contents of the file system to an earlier version. To handle merge conflicts, CodingSpectator has built-in support for automatic conflict detection and resolution. When CodingSpectator detects a merge conflict, it removes the developer's data from the central repository and then submits the new data. Despite removing the data from the central repository, it is possible to locate the merge conflicts and restore the data that was collected before the conflicts occurred.

CodingSpectator prompts the developers for their Subversion user names and passwords when CodingSpectator is about to submit their data. CodingSpectator gives the developers the option to save their passwords in Eclipse securely. See `http://codingspectator.cs.illinois.edu/documentation` for more information about the features of CodingSpectator for developers.

2. **Setting Up an Eclipse Update Site -** Users of CodingSpectator install it from an Eclipse update site[18]. An Eclipse update site is an online repository of the JAR and configuration files that Eclipse requires for installing a plug-in.

   You will have to customize CodingSpectator at least by specifying the URL of the Subversion repository to which CodingSpectator should submit developers' data. You may also want to customize the message that CodingSpectator shows to the developers when it prompts them for their Subversion credentials. You can customize these aspects of CodingSpectator by changing the configuration files that are packed in the existing JAR files hosted at the Eclipse update site of CodingSpectator. If you need to customize CodingSpectator in more complex ways that involve changes to its source code, you should follow the instructions for building CodingSpectator's update site from source code.

### *3.3.3. Extending CodingSpectator*

In addition to collecting detailed refactoring data, CodingSpectator provides a reusable infrastructure for collecting Eclipse usage data. Extending CodingSpectator frees you from having to develop many features from scratch, e.g.,, Subversion communications, automatic merge conflict detection and resolution, secure storage of Subversion credentials, and periodic update reminders.

CodingSpectator provides an Eclipse extension point (id = `edu.illinois.codingspectator.monitor.core.submitter`) and the following interface:

```
1 public interface SubmitterListener {
2   // hook before svn add
3   void preSubmit();
4   // hook after svn add and before svn commit
5   void preCommit();
6   // hook after svn commit
7   void postSubmit(boolean succeeded);
8 }
```

[17] http://svnkit.com/

[18] `http://codingspectator.cs.illinois.edu/installation`

The above interface provides three hooks to CodingSpectator's submission process. CodingSpectator checks out the Subversion repository into a folder, which we refer to as the *watched folder*. Then, it executes the Subversion commands (e.g.,, add and commit) on the watched folder. A plug-in that extends the `submitter` extension point and implements the `SubmitterListener` interface can perform actions before or after two of the Subversion commands that CodingSpectator executes: add and commit. For example, CodingSpectator overrides the method `preSubmit` to copy the recorded refactoring descriptors to the watched folder. As another example, the developers of CodingSpectator made the Eclipse UDC plug-in use the `submitter` extension point and copy the UDC data to the watched folder. As a result, CodingSpectator submits the UDC data to the Subversion repository. Effectively, this is an alternative method to the one presented in Section 3.1.3 for collecting UDC data in a central repository.

### *3.4. Build it Yourself for Visual Studio*

This section shows how to implement a usage data collection tool for Visual Studio that generates the Navigation Ratio metric (see Section 2.2) daily, giving the developer insight into his own navigation patterns. Readers attempting the tutorial should be familiar with C# as well as have a working knowledge of Visual Studio.

Because this extension is illustrative some simplifications have been made that would need to be addressed in a widely deployed extension. For instance, this example extension does not perform any background processing, thus the developer may notice a delay during Visual Studio start-up.

#### *3.4.1. Creating a Visual Studio Extension*

1. **Create a new extension solution** - With the Visual Studio SDK installed, create a new Visual Studio Extension project with a project name of "Collector" and a solution named "VisualStudioMonitor". Setup the extension to provide a menu command a named "Stop Monitoring" with a command ID of "StopMonitoring". To separate the Visual Studio Extension setup code from the core functionality of the extension, create a second project within the same solution called "Monitor".

2. **Ensure the extension loads on start-up** - The next step is to instruct the extension package to load when Visual Studio starts, by setting the attribute ProvideAutoLoad on the package class (CollectorPackage.cs). The GUID value in the below listing will load the package when Visual Studio starts.

```
1  // This attribute starts the package when Visual Studio starts
2  [ProvideAutoLoad("{ADFC4E64-0397-11D1-9F4E-00A0C911004F}")]
3  [Guid(GuidList.guidCollectorPkgString)]
4  public sealed class CollectorPackage : Package
```

3. **Create the Monitor project** - Add a class library type project to the "VisualStudioMonitor" solution. Because the class library must be signed, go to the Properties for the Monitor project and select Signing from the list at the right. In the Signing tab, check the "sign the assembly" check-box then under "Choose a strong name key file", select Browse and browse over to the Key.snk file in the collector project (the file was created with the solution).

4. **Create the monitoring class** - The next step is to create a static class that will manage the log file including starting, stopping recording data, and inserting data into the log file. Rename the class created by Visual Studio in the Monitor project to "DataRecorder". Because we don't want more than one recorder running at a time and want to access this class without instantiating it, make the class static. Create a method to Start the recorder that generates a file name for the log file and sets a flag that the recording has started. A Stop method resets that flag and perhaps clears the file name. A method to write a log message to the file completes DataRecorder.

5. **Connecting the extension framework with the recorder** - Finally, insert a call to DataRecorder.Start() at the end of the Initialize() method in the CollectorPackage class. This will start the monitoring

each time Visual Studio starts. You will need to add a reference for the "Monitor" project to the Collector project, make sure you sign the Monitor project, then rebuild the solution.

See the listing for the CollectorPackage.cs in Listing 1 and DataRecorder.cs in Listing 2 in the listings at the end of the chapter .

### *3.4.2. Create the Data Model*

The next step creates a data model for storing and managing event monitoring for Visual Studio. This includes designing the main event types and implementing a factory to create these events.

1. **Implement the base class** - Create the AbstractMonitoredEvent class in the Monitor project in Visual Studio. Then add properties for EventName and Classification as follows.

```
1   [XmlInclude(typeof(MonitoredCommandEvent))]
2   [XmlRoot(ElementName = "MonitoredEvent", Namespace = "http://Monitor")]
3   public abstract class AbstractMonitoredEvent
4   {
5       /// <summary>
6       /// Default constructor to use in serialization
7       /// </summary>
8       protected AbstractMonitoredEvent()
9       {
10      }
11
12      public String EventName { get; set; }
13      public String Classification { get; set; }
14  }
```

2. **Enable serialization in base class** - So that we can store events in a configuration file then manipulate that configuration file later, we provision this abstract class for XML serialization of itself and its derived classes. Dot NET attributes support the XML serialization in this structure. The first attribute tells XML serialization that the MonitoredCommandEvent class is a derived class of AbstractMonitoredEvent that we will create next. This provides the ability to serialize and de-serialize the public objects of the derived class by referencing the type of AbstractMonitoredEvent when creating a serializer. The second attribute creates an XML name-space that all derived classes will share with the AbstractMonitoredEvent class.

3. **Create the concrete subclass** - The next step is to create a derived class called MonitoredCommandEvent that inherits from AbstractMonitoredEvent. MonitoredCommandEvent implements a constructor that builds a MonitoredCommandEvent object from the Command class of the DTE.

   The EnvDTE.Command object contains fields for Guid (a GUID string), ID and integer sub-id, and Name a readable name for the command. To register an event handler for a EnvDTE.Command, you need get an object reference for the command using the Guid and ID to identify the command. The GUID is a Globally Unique IDentifier for command events in Visual Studio, however, some command events share a GUID and distinguish themselves with different EventIDs. Thus both elements are necessary to link a Command event from the DTE to an event handler in this extension. The Name is useful information to understand what the command is. There are several versions of the DTE object corresponding to versions of Visual Studio. Depending on the commands of interest, each version may need to be queried for its commands.

   The constructor that takes a Command as input, simply extracts the necessary and relevant fields from the DTE's Command object and transfers the matching information into the corresponding fields from this class and the AbstractMonitoredEvent class.

4. **Enable serialization in concrete subclass** - Ensure the class also includes a constructor that builds from an XElement and an output method ToXElement translates the object to XML for saving. Add using statements for System.Xml.Serialization, and EnvDTE and their corresponding references in the project References configuration.

```
[XmlRoot(ElementName = "MonitoredEvent", Namespace = "http://Monitor")]
public class MonitoredCommandEvent : AbstractMonitoredEvent {

     public int EventID { get; set; }
    public String Guid { get; set; }

    public MonitoredCommandEvent()
    {
    }

    public MonitoredCommandEvent(Command DTECommandObj) {
        if (DTECommandObj != null) {
            this.EventName = DTECommandObj.Name;
            this.Classification = EventName.Split('.')[0]; //use the first part of event name
            this.Guid = DTECommandObj.Guid;
            this.EventID = DTECommandObj.ID;
        }
        else {
            throw new ArgumentNullException("DTECommandObj");
        }
    }
```

The attribute for XMLRoot is the same attribute assigned to the AbstractMonitoredEvent class which tells XML Serialization that this type is a type belonging to the abstract class. In this class, create two public fields, EventID as int and GUID as string, that will save important information from the Visual Studio DTE object needed to engage monitoring for each command.

5. **Create the event factory** To complete the Simple Factory pattern, a static factory class provides static factory methods that creates an object of type MonitoredCommandEvent from a DTE Command object and returns it as an AbstractMonitoredEvent. For now the only class to consider is the MonitoredCommandEvent derived class, however, a future step will add more derived classes.

### *3.4.3. Storing Visual Studio Command Event Info*

Our extension is now wired to listen for events, however, events also need to be saved for later analysis. In this step we discuss how the data is collected and persisted.

1. **Create the collection manager class** - In this step, build the MonitoredEventCollection class shown in Listing 6 that manages a List object of AbstractMonitoredEvent type.

2. **Create and populate the configuration** - Configuration data is stored in the List object. The List object gets populated from an XML file that stores the configuration data. The MonitoredEventCollection class provides a method to query the DTE for all commands and initialize the list. Another method called after the DTE query stores the List contents in the same XML format file. These two methods should be called in sequence the first time the extension launches. After that, it reads the XML file on start-up to initialize the List. Call the method(s) to query, store and load the event list from the Start() method of the DataRecorder class in the previous step so that the Monitor will load the commands on start-up.

   Fortunately the DTE object has a query method that lists all the commands it manages. The DTE Commands object returns an IEnumerable collection of EnvDTE.Command objects. The listing below provides a method to try to get an instance of the DTE. It depends on references to EnvDTE, Microsoft.VisualStudio.Shell.12.0, and Microsoft.VisualStudio.OLE.Interop so be sure to add those to the project's References list.

```

    using EnvDTE;
    using Microsoft.VisualStudio.Shell; //12.0
    private static DTE tryGetDTEObject()
    {
```

```
        DTE dteobj=null;
        try
        {
            dteobj =
                ((EnvDTE.DTE)ServiceProvider.GlobalProvider.GetService(typeof(EnvDTE.DTE).GUID)).DTE;

        }
        //Important to catch the following exception if the DTE object is unavailable
        catch (System.Runtime.InteropServices.InvalidComObjectException)
        {}
        //Important to catch the following exception if the DTE object is busy
        catch (System.Runtime.InteropServices.COMException)
        {}
        return dteobj;
    }
```

Once you have a reference to the DTE object from the tryGetDTEObject method, use the DTE to query the Commands object. Then process each command into the List managed by MonitoredEventCollection. Example code from QueryVSForAddDTECommands method in MonitoredEventCollection.cs in Listing 6 is highlighted below making use of the MonitoredEventFactory to generate each AbstractMonitoredEvent stored in the List. The try-catch here is necessary because the saved DTE object could get disposed while the loop processes the Commands.

```
            try
            {
                foreach (Command DTE_CommandEventObj in dteobj.Commands)
                {
                    AbstractMonitoredEvent NewEvent =
                        MonitoredEventFactory.GetMonitoredEvent(DTE_CommandEventObj);
                    if (NewEvent != null)
                    {
                        EventList.Add(NewEvent);
                    }
                }
            }
            //This exception happens during dispose/finalize when VS exits, just return null
            catch (System.Runtime.InteropServices.InvalidComObjectException)
            {
                return null;
            }
```

3. **Enable persistence of the configuration** - A persistent configuration file helps independently manage the events that get monitored for a study, and makes the configuration of all possible events easier to manage. Using the framework's ToXelement methods, build methods in MonitoredEventCollection to save the List of AbstractMonitoredEvents to the configuration file and load them from the configuration file. Below is the core code for the saveEventInterestTable method in MonitoredEventCollection.cs in Listing 6 that creates an XML serializer for the List object then writes that to the file stream.

```
            var serializer = new
                System.Xml.Serialization.XmlSerializer(typeof(List<AbstractMonitoredEvent>));
            using (Stream file = new FileStream(filepath, FileMode.Create, FileAccess.Write))
            {
                serializer.Serialize(file, eventList);
                file.Flush();
            }
```

*3.4.4. Register Event Handlers*

Now that the framework is complete and a configuration file for all command events to be monitored is ready, the methods to hook Visual Studio into event handlers that log each command can be created. This step will add methods and member objects to AbstractMonitoredEvent and MonitoredCommandEvent classes to register event handlers with the DTE and dispose of them appropriately when necessary. The MonitoredEventCollection class gets a new method to perform the registration from the objects in the list and another method to de-register them.

1. **Define the registration interface** - The AbstractMonitoredEvent class should get a virtual RegisterEventForMonitoring method that takes an object parameter we will use to pass a DTE reference in. The method returns a bool based on successful registration. The class also gets a non virtual Dispose() method and a virtual Dispose(bool disposing) method with the former calling the latter and the latter setting a field isDisposed to true. This is the typical dispose structure. Finally, the abstract class holds the non-virtual method to write the event log information (the abstract class's fields and a time-stamp) to the log via the DataRecorder class. This unites logs for all derived classes into a common format.

2. **Implement the registration routine** - The MonitoredCommandEvent in Listing 4 class overrides the virtual RegisterEventForMonitoring method to implement registering an event handler for Command events. Registering first must find the event in the DTE and assign it to the field, then attach a new event hander to the event. Looking at the method listing below, the Guid and EventID are used as parameters to query the DTE Events object for the specific command event in this instance. The result is assigned to a field, eventTypeObject. With this reference to the event, the next block adds an event handler that runs after the command is executed. After all that if the eventTypeObject is not null, the method returns true for success.

```
1         public override bool RegisterEventForMonitoring(object dte)
2         {
3             if (!isDisposed && eventTypeObject == null && dte != null)
4             {
5                 eventTypeObject = (dte as DTE).Events.get_CommandEvents(Guid, EventID) as
                      CommandEvents;
6             }
7             if (eventTypeObject != null)
8             {
9                 eventTypeObject.AfterExecute += new
10          _dispCommandEvents_AfterExecuteEventHandler(OnAfterExecute);
11            }
12            return (eventTypeObject != null);
13        }
```

With the above method in Visual Studio, the missing fields and methods can be auto-generated via the "Generate" context menu command.

The last step with MonitoredCommandEvent is to create the Dispose method that will de-register the event handler. This looks like the following:

```
1         protected override void Dispose(bool disposing)
2         {
3             if (eventTypeObject != null)
4         eventTypeObject.AfterExecute -= OnAfterExecute;
5             this.isDisposed = true;
6         }
```

Use the Visual Studio "Generate" command to generate a method stub for OnAfterExecute and the code will compile. In the OnAfterExecute method, call ToLog so the event data is captured in the log.

3. **Register all commands** - MonitoredEventCollection in Listing 6 now needs methods to perform registration and de-registration on all the events in the list. As the following listing shows, RegisterEventInventoryForEventMonitoring() must get the DTE object then walk through the IDEEventListenerRegistry list calling the abstract method RegisterEventForMonitoring with the DTE. If one of them succeeds then this method considers it successful.

```
1        public bool RegisterEventInventoryForEventMonitoring()
2        {
3            DTE dteobj = tryGetDTEObject();
4            bool somethingRegistered = false;
5            if (dteobj != null && IDEEventListenerRegistry != null && IDEEventListenerRegistry.Count > 0)
6            {
7                foreach (AbstractMonitoredEvent command in IDEEventListenerRegistry)
8                {
9                    if (command.RegisterEventForMonitoring(dteobj))
10                   {
11                       somethingRegistered = true;
12                   }
13               }
14           }
15           return somethingRegistered;
16       }
```

4. **Connect to the package life-cycle** - Refactor the MonitoredEventCollection object in DataRecorder to a static class field. Then add a call to RegisterEventInventoryForEventMonitoring() in the Start() method of DataRecorder. Add a call to the de-register method of MonitoredEventCollection in the Stop() method of DataRecorder.

5. **Execute the extension** - Run the solution and use a few commands in Visual Studio, then give the Stop Collector command and check the log file. You should see output like the following:

```
Collector Started
2014-02-02 13:46:52Z,Tools.AddinManager,Tools
2014-02-02 13:46:56Z,Tools.ExtensionsandUpdates,Tools
Collector Stopped
```

Below are descriptions of the code listings for the example code we discussed in this section. Code listings are located at the end of the chapter.

- The Listings for AbstractMonitoredEvent.cs in Listing 3 shows the additions to that class.
- The listing for CommandMonitoredEvent.cs in Listing 4 shows methods implemented for registration and disposal.
- The listing for MonitoredEventCollection.cs in Listing 6 shows list processing in calls to respective registration and de-registration methods for the List object.
- The DataRecorder class is shown in Listing 2.

With this demonstration you see how to build a usage monitor for Visual Studio that records most commands the developer can issue in the IDE. What's missing? Well there are other areas of the DTE to explore such as editor events, unit test events, build and debug session events that provide greater context to the developer experience. For brevity, capturing those events is left to the reader to explore on their own.

Thus far we have been focusing on concrete usage data collection frameworks and the specific data collected by these frameworks. With options to collect data from both Visual Studio and Eclipse, we hopefully provided a good resource to get you started towards collecting and analyzing usage data. Next, let's look next at methods and challenges in analyzing usage data.

## 4. How to Analyze Usage Data

The tools described in this chapter provide the usage data you can use to study developer interactions, but leave the selection of methods for analyzing the data to the reader. In this section we discuss several data analysis techniques that you may apply to usage data. We will discuss attributes of the data including format and anonymity of the data, categorizing the records, defining sequences, using state models, and other techniques to extract information from the usage data.

### *4.1. Data Anonymity*

**Non-anonymous data**, where sensitive information including source code snippets, change-sets, and even access to the source code base is provided has obvious advantages. We can replay developers' activity stream, affording them a deep understanding of the developer's actions e.g. (Vakilian et al., 2012). There are few limits on how this data can be analyzed, and *non-anonymous data is well-suited for exploratory studies*. Unfortunately, there are some key disadvantages. First, it may not be permitted for developers to participate in a study; typical enterprise developers may face termination if they were to leak even parts of their source code base. Second, while playback and other deep analyses are possible, these analyses can be costly in terms of time and resources.

**Anonymous data**, where only records of activities and anonymous facts about artifacts are recorded, may at first seem strictly inferior. Indeed there are some limitations on what can be learned from anonymous activity streams, yet there are key advantages. First, Snipes et al. (2014) report that developers are more receptive to sharing anonymous data, and thus the ability to collect a large amount of information from many developers increases greatly. Second, because the data set is relatively large and is harvested from working developers, conclusions are ultimately more reliable.

In this section we focus on analyzing anonymous data sources. We do so because analyzing anonymous activity streams is similar to analyzing non-anonymous data streams (i.e., they are both activity streams) and because the unlimited variation of analysis permitted by non-anonymous data affords few generalities. As we discuss analyzing usage data we start with straightforward magnitude analysis, build to a categorization system for activity streams, discuss dividing streams into sessions, and finally state-based analysis.

### *4.2. Usage Data Format*

Most usage data is collected as an activity stream with varying levels of supporting detail. In Figure 9 we present an abstraction of a typical activity stream. It includes a time-stamp, followed by the activity, appended with (often anonymous) details concerning that activity. We can apply this model to the examples discussed earlier. For instance, the event recorded by UDC corresponds to a row in our theoretical model. It includes a time-stamp (i.e., time), an activity description (i.e., what,kind,and description), and additional information (i.e., bundleId, bundleVersion). Similarly, the CodingSpectator example includes a time-stamp (i.e., stamp), an activity description (i.e., id), and a much larger set of additional information (i.e., code-snippet, selection, selection-in-code-snippet, etc.). Because these and other usage data activity streams can easily be described using our abstraction we will refer to it as we describe data analysis techniques.

### *4.3. Magnitude Analysis*

A major advantage of anonymous usage data is the fact that it captures developers in their natural habitat, without any observational bias. Deriving conclusions from hours of developers' field work is naturally more convincing than from hour-long, in-laboratory developer studies. One type of questions that usage data is well-suited to answer uses measurement of the magnitude of occurrence of a specific event. For instance, we may want to know "How often do developers invoke the pull-up refactoring" or "How often is the file search invoked?" By performing a count of a specific message in the collected logs, researchers can easily calculate frequencies of specific actions that can be often sufficient to answer important questions.

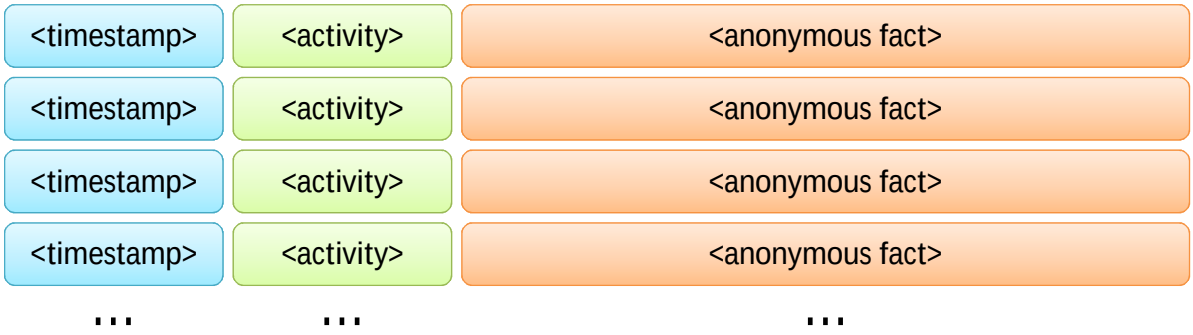


Figure 9: Abstract model of developer activity streams.

However, there are a few common issues with magnitude analysis. First, in any sufficiently large set of user logs there is a small set of developers that will use the feature/tool under analysis orders of magnitude more often than the general population, potentially skewing the data. Second, attempts to attribute time to individual activities are fraught with difficulties. For instance, there is a temptation to report the percentage of time spent doing activity X. Yet, because the data is based on a stream of activities any time calculation requires making unsound assumptions on what happens in between these events.

Murphy et al's Murphy et al. (2006) work on understanding Eclipse IDE usage provides several examples of magnitude analysis being used effectively. By simply counting instances of events in developers' activity streams they were able to present how often developers accessed certain views, the top 10 commands executed by developers, and the percentage of developers that used each specific refactoring command. In spite of the simplicity of this analysis its ideal for identifying heavily used features for improvements and unused features for removal, as well as for getting a sense of how developers are currently working.

### *4.4. Categorization Analysis*

While magnitude analysis is well-suited for answering questions about the use of a single IDE command, many questions are related to a specific feature or tool in the IDE, which usually maps to multiple activities. For instance, the question "How often are refactorings performed?" cannot be answered via magnitude analysis alone, as refactorings can be triggered through a number of different IDE commands. These commands first need to be categorized, after which magnitude analysis can be used.

When focusing on a concrete sub-task, such as refactoring, it may be easy to categorize activities. In this case, all refactoring commands, such as pull-up or extract method, can be classified as refactorings. However, when focusing on more general behavior, such as editing, navigating, and searching, categorization can be difficult. It is impossible to say, for instance, from a single click in the `File Explorer` window, whether that click represents a search, as the developers browses a few promising files, or a navigation, as he implicitly opens a type declaration of a variable he was just browsing. Thus, categorization without context can produce noisy data in certain cases. However, categorization is a powerful tool, especially when there is little ambiguity in the IDE commands that are analyzed.

To illustrate both the power and limitations of category analysis consider the following IDE data stream, asking the question "Do developers use code search tools?".

For this question, the category of log events related to code search tools should be identified and counted. Modern IDEs commonly offer several code search tools, which operate at the global or local scale, such as the `Find-in-Files` and `Quick Find` tools. An example log from Visual Studio with these tools is shown in Figure 10 based on Visual Studio. Using categorization analysis we can identify three log events related to usage of code search tools, and report various statistics aimed at answering the question (e.g., number of code search events per day, number of code search events per developer see Figure 11). However, the IDE usage data can sometimes be affected by noise, which cannot be avoided by

```
Collector Started
2014-02-02 13:21:22 - User submitted query to Find-in-Files
2014-02-02 13:24:36 - Find-in-Files retrieved 42 results
2014-02-02 13:32:21 - User clicked on result 2
2014-02-02 13:46:56 - User submitted query to Quick Find
2014-02-02 14:07:12 - Open definition command; input=class
2014-02-02 14:46:52 - User submitted query to Find-in-Files
2014-02-02 14:46:56 - Find-in-Files retrieved 8 results
2014-02-02 14:48:02 - Click on File Explorer
```

Figure 10: Log File Search Category Example

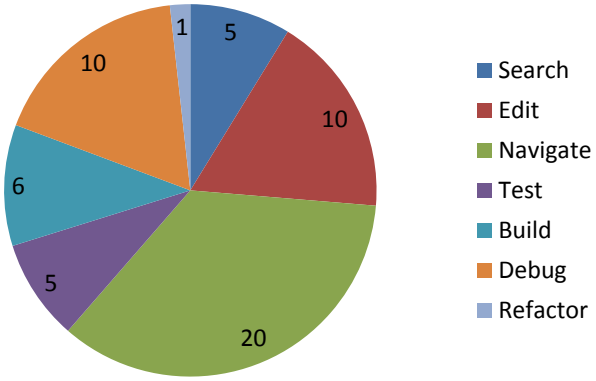


Figure 11: Categorized Log Events with Search Category

categorization analysis. For instance, the second query to `Find-in-Files` is not followed by a developer click, which is a failed interaction with the `Find-in-Files` tool and should likely not be included in answering the question.

### *4.5. Sequence Analysis*

Magnitude analysis and categorization are both appropriate for answering questions that are simply manifested in the IDE usage log. However, a more powerful way of analyzing activity logs is through sequence analysis, which first breaks the IDE data stream into a number of sequences, according to some criteria, and then reports upon characteristics of each sequence. A sequence in the IDE usage data corresponds to a software engineering task or sub-task accomplished by the developer (e.g. refactoring, looking for a starting point for a maintenance task, etc.), consisting of all of IDE events in a given time span. For instance, answering the question of "Are developers successful at finding initial points in the code for a software maintenance task?" requires that the sequence of IDE events corresponding to each maintenance task be identified, before we can perform further analysis using either magnitude or categorization analysis. The granularity of a sequence is determined by the guiding question. For certain questions, we may be interested in a smaller sequences (e.g., searching the code base), while for others we may need to consider a longer time span (e.g., implementing a new feature, fixing a bug). According to Zou and Godfrey (2012), the larger and more complex the task or sub-task to extract, the harder it is for sequence analysis to determine its starting and stopping point in the activity log.

In many cases, extracting activity sequences can be challenging as it impossible to know exactly when a developer begins or ends a particular task or sub-task, without understanding the developer's underlying thought process. There are several possibilities in how sequence extraction can be performed, based on the specific question. One possibility is to use sentinels, which are specific actions that indicate

```
Collector Started
2014-02-02 13:46:52 - User submitted query to Find-in-Files
2014-02-02 13:46:56 - Find-in-Files retrieved 121 results
2014-02-02 13:52:21 - User clicked on result 2
2014-02-02 13:58:01 - User clicked on result 8
2014-02-02 13:59:57 - Open caller/callee command
...
2014-02-02 14:46:52 - User submitted query to Find-in-Files
2014-02-02 14:46:56 - Find-in-Files retrieved 19 results
2014-02-02 15:01:08 - User clicked on result 11
2014-02-02 17:30:12 - User edited code
...
```

Figure 12: Log File Sequence Example

the beginning and end of a sequence. For instance, in the context of the code search question mentioned above, submitting a query to the code search tool begins a sequence, while performing an edit or structural navigation (e.g., following the call graph) ends the sequence. Another possibility is to use the passage of time to extract sequences, where time without any activity is used as a signal of task start or finish. Yet another possibility is to use locations in the code base to identify a sequence in the activity log. This is the key insight used in Coman and Sillitti (2008)'s algorithm, which uses periods of activity on the same program elements to represent the core of a task, and time distance between such events to extract sequences corresponding to developer tasks. In laboratory validation studies of this algorithm have shown very high accuracy (80%) when compared to the ground truth reported by developers. Zou and Godfrey (2012) find that this accuracy may not hold up in an industrial setting, where tasks are longer, code bases are more complex, and developer interruptions are common. Also, the algorithm requires that information regarding program elements is kept in the activity log, which may conflict with many developer's privacy and anonymity requirements.

We illustrate sequence analysis with the usage log shown in Figure 12 and the question "Are developers successful at finding initial points in the code for a software maintenance task?" To answer the question, sequence analysis can extract two separate sequences in the log from Figure 12 by using sentinels indicative of start/end of a code search activity, among other possible sequence identification approaches. Both of the extracted search sequences can be deemed as successful since for each of the queries to the `Find-in-Files` code search tool the developer clicks on a retrieved result, followed by a structured navigation event (open caller/callee) or editing event (developer edited code). Therefore it would seem that the developer is successful at finding initial points in the code for his or her software maintenance task. However, on closer inspection of the second sequence, we observe that there is a large time gap between clicking on a result and editing code. The reason for this time gap is open to interpretation, as the developer may have returned to the previous results, continuing the search session, or have started on a completely new development activity. Certain types of sequence analysis, such as Coman's algorithm, take time into account when identifying sequences, while others, like the sentinel approach used above, do not. Neither of these approaches, however, helps to resolve the origin of ambiguous events, which are left to the reader to characterize.

### *4.6. State Model Analysis*

Another way to analyze log data is to view the log as a sequence of events occurring in a state machine. Using state models, we can quantify the occurrences of repeating actions and describe a usage pattern in statistical analysis. **?** used sequence analysis to generate a graphical view of a profile of how users interacted with the components of a system from log data. In state model analysis, the sequential data is converted to nodes and edges of a graph which represents the entire data in states and transitions

```
2013-03-21 18:18:32Z,A
2013-03-21 18:18:33Z,B
2013-03-21 18:20:49Z,C
2013-03-21 18:20:50Z,A
2013-03-21 18:20:56Z,B
2013-03-21 18:20:57Z,A
2013-03-21 18:21:08Z,C
2013-03-21 18:21:08Z,D
2013-03-21 18:21:08Z,E
2013-03-21 18:21:08Z,A
```

Figure 13: Sample Log File to Convert to State Model

between them. A Markov state model provides information about the probability of occurrence of each state and the transitional probability of each activity. The statistics provided in a Markov state model include the quantity of time and probability for the developer being in each state. From a given state, the model calculates the probability of each transition to different unique states. State models answer specific questions such as what is the probability that once a developer searches the code, they edit a code element listed in the find results. Expanding this question, the probability of an entire use case or set of transitions through the usage model, is calculable from the state model.

State model graphs make it easy to identify the most active states and edges provides information about the important activities in the data set. As the number of states increases, the Weighted Directed Graph (WDG) becomes more complex and hence difficult to understand. When this occurs, summarizing the detailed event data into higher-level categories effectively combines the states to get more meaningful information. For example, classifying events in usage data of similar types into categories results in fewer states with the same number of transitions as the original data.

We generate a state model from a usage log by transforming the serially ordered events in a sequence data to a WDG data structure. We can group the information in the log line to any level, such as, event level, event category level, tool level, or application level. In the sequence data, each event is important as a standalone event, however, in the WDG representation, the importance shifts to adjacent pairs of events. Therefore, each group in the sequence data is represented by a unique node in the WDG. For example, suppose we choose to create a WDG at the level of an event. The analysis that generates the graph creates a node for each unique event name, then determines the transitions that occur in the log before and after that event name.

To understand how to interpret a state model, look at our example graph Figure 14. We see that an edge exists from one node (head) to another (tail) if there is an occurrence of the event representing the tail node immediately after the event representing the head node in the log file. For example if event B follows event A in the log file, then there is directed edge from node A to node B in the WDG. The edges are labeled with the number of times this transition has occurred. For example if B occurs a fifty times after A in the log file, then the edge from node A to node B in the WDG is labeled with 50. Typically, we keep track of the actual count as the weight of the edge, when building the graph. In the graph we display the percentage value as the label. This percentage is proportional to the total number of transitions. The cumulative probability of out-edges of a node is 1. The transitional probabilities of each out-edge is calculated by dividing the number of transactions of that out-edge with the sum of all outward transactions from that node. We could also store the dynamic parameter information in each log line as a list along the edges.

As an example, consider a sample log file shown in Figure 13 given below and the corresponding state model graph in Figure 14. Intuitively you can see how the state model represents the log states and transitions between them in the sequence of events.

Converting a log into a state model requires three steps. We use JUMBL (Java Usage Model Builder)

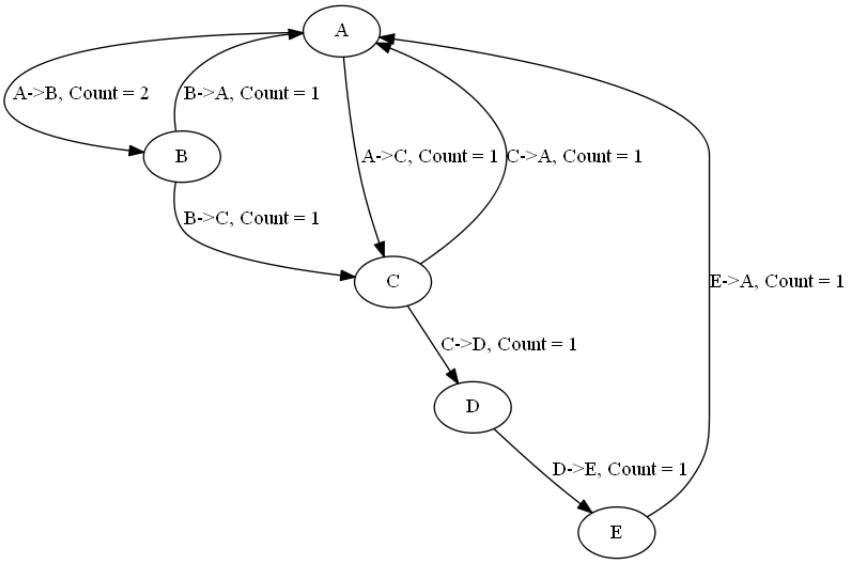


Figure 14: Weighted Directed Graph of the Example Log

from the Software Quality Research Laboratory (SQRL) of the University of Tennessee. Details on input formats and JUMBL tools are available on sourceforge.[19]

1. First, convert the log file into a representation for each transition called a sequence based specification. The format for a sequence based specficicaiton in csv files is described in the JUMBL user guide. This representation contains the following information with one row for each transition :
   - State transition
   - Count of transitions
   - Total time elapsed
   - State In information
   - State Out information
2. After importing the sequence based spec, JUMBL can write out the representation of a state model as a TML script or several other formats including GML (Graph Modeling Language) that graph tools can import. The TML has information about the nodes, the out edges from each node along with the number of transitions from each node to another. The corresponding graph for the usage log example is depicted in Figure 14

Using state models, a sequence data with hundreds of thousands of lines can be quickly converted to more meaningful graphical representation using this method. Once the TML file is generated, we can use JUMBL to find out the state probabilities of each states. Using the state probability and the usage patterns we can draw conclusions about the occupancy of individual states and the use cases that involve transitions through several states.

*4.7. The Critical Incident Technique (CIT)*

The Critical Incident Technique (CIT) is a general methodology for improving a process or system with respect to a set of objectives. CIT prescribes a systematic study of the *critical incidents*. Critical incidents are *positive* or *negative* events that have a significant effect on the objectives.

[19] `http://jumbl.sourceforge.net/`

CIT was developed and published in its current form by Flanagan in 1954 (Flanagan, 1954). Nevertheless, it is believed that the technique was introduced even earlier by Galton (circa 1930). Variations of CIT have been widely used in Human Factors (Shattuck and Woods, 1994).

del Galdo et al. (1986) applied CIT to Human-Computer Interaction (HCI) as part of evaluating the documentation of a conferencing system. They asked the study participants to perform a task and report any incident that they ran into. The researchers observed the participants during the study, analyzed the reported incidents, and proposed improvements to the documentation accordingly.

In the context of IDEs, Vakilian and Johnson (2014) adapted CIT to automated refactorings. The goal of this study was to identify the usability problems of automated refactorings by analyzing refactoring usage data. The researchers found that certain events such as cancellations, reported messages, and repeated invocations are likely indicators of the usability problems of automated refactorings. By locating these events in the usage data and analyzing their nearby events, the researchers were able to identify 15 usability problems of the Eclipse refactoring tool. For instance, the usage data indicated that six participants invoked the Move Instance Method refactoring for a total of 16 times but none finished the refactoring successfully. In all cases, the participants either canceled the refactoring or could not proceed because of an error that the refactoring tool reported. By inspecting these critical incidents, the researchers were able to infer two usability problems related to Move Instance Method.

To apply CIT on a set of usage data, you should follow several steps. First, identify the objectives. Finding usability problems is only one example. Second, identify a set of events that may be critical incidents. These are events they may have significant effects on your objectives. Usually, the negative critical incidents, which may have negative effects on the objectives, are better indicators of problems than the positive ones. Third, identify the critical incidents in your usage data. Fourth, collect sufficient contextual information to interpret the critical incidents. This may include events that have occurred in close time proximity to the critical incidents. You may even have to interview the developers or ask them to report their explanations of the incidents during the study. Although the developer reports may be short or incomplete, they can provide more insight about the nature of the problems. Finally, evaluate the effectiveness of the critical incidents and revisit the steps above. The process that we described for applying CIT on usage data is iterative in nature. That is, it is natural to go back to a previous step from each step.

### *4.8. Including Data From Other Sources*

Other data sources, such as developer feedback, task descriptions, change histories can provide context for usage data and yield new opportunities for supporting developers. In this section, we briefly outline some related work in this area.

#### *4.8.1. Developer Feedback*

Usage data collection can be augmented by explicit developer feedback to establish a ground truth. For instance, if you want to understand a developer's opinion of a tool or his or her purpose for using an IDE feature, it is best to ask about it. A post-analysis interview can shed light on your observations and confirm your analysis or point out different aspects.

Information on a developer's activities can provide an additional rational to the usage data and explain the developer's actions. Asking questions on how well the developer knows the code he or she is working on can explain a developer's navigation patterns and usage of code search e.g. (Snipes et al., 2014), while questions on the task, e.g., implementing a new feature versus fixing a bug, can explain other characteristics, such as the amount of editing versus testing.

#### *4.8.2. Tasks*

As mentioned above, Mylin is a popular extension to the Eclipse IDE that supports task management, reducing the effort for a developer to switch between tasks and maintaining relevant information for each task (Kersten and Murphy, 2006). In the context of a specific task, Mylin collects usage data in order to compute a degree-of-interest (DOI) value for each program element, which represents the interest the developer has in the program element for the task at hand. Program elements that a developer

interacted with frequently and recently have a higher DOI value. Using these calculated DOI values, Mylyn highlights the elements relevant for the current task and filters unnecessary information from common views in the IDE.

*4.8.3. Change History*

The FastDash tool by Biehl et al. (2007) enables real-time awareness of other developers' actions (e.g., focus or edits to a specific file) by utilizing usage data in concert with the source files and directories. FastDash's purpose is to reduce faults caused by lack of communication and lack of awareness of activities that other developers are performing on the same code base. The tool highlights other developer's activity as it is occurring using a sophisticated dashboard on the developer's screen or on a team's dashboard.

Using this overview of analysis techniques should give you a good start towards analyzing usage data. The ideas you bring to your usage data analysis process provide the real opportunities for innovating usage data based research. Next we will discuss limitations we have observed collecting and analyzing usage data.

## 5. Limits of What You Can Learn from Usage Data

Collecting usage data can have many interesting and impactful applications. Nonetheless, there are limits to what you can learn from usage data. In our experience, people have high expectations about what they can learn from usage data, and those expectations often come crashing down after significant effort implementing and deploying a data collection system. So before you begin your usage data collection and analysis, consider the following two limitations of usage data.

**Rationale is Hard to Capture.** Usage data tells you what a software developer did, but not why she did it. For example, if your usage data tells you that a developer used new refactoring tool for the first time, from a trace alone you cannot determine whether (a) she learned about the tool for the first time, (b) she had used the tool earlier, but before you started collecting data, or (c) her finger slipped and she pressed a hotkey by accident. We do not know whether she is satisfied with the the tool and will use it again in future. It may be possible to distinguish between these by collecting additional information, like asking the developer just after she used the tool why she used it, but it is impossible to definitively separate these based on the usage data alone.

**Usage Data Does Not Capture Everything.** This is practically impossible to capture "everything," or at least everything that a developer does, so using the goal-question metric methodology helps narrow the scope of data required. If you have a system that captures all key presses, you are still lacking information about mouse movements. If you have a system that also captures mouse movements, you're still missing the files that the developer is working with. If your system captures also the files, you're still lacking the histories of those files. And so on. In a theoretical sense, one could probably collect all observable information about a programmer's behavior, yet the magnitude of effort required to do so would be enormous. Furthermore, a significant amount of important information is probably not observable, such as rationale and intent. Ultimately, usage data is all about fitness for purpose – is the data you are analyzing fit for the purpose of your questions?

To avoid these limitations, we recommend thinking systemically about how the data will be used while planning rather than thinking about usage data collection abstractly. Invent an ideal usage data trace, and ask yourself:

- Does the data support my hypothesis?
- Are there alternative hypotheses that the data would support?
- Do I need additional data sources?
- Can the data be feasibly generated?

Answering these questions will help you determine whether you can sidestep the limitations of collecting and analyzing usage data.

## 6. Conclusion

Analyzing IDE usage data provides insights into many developer activities which help identify ways we can improve software engineering productivity. For example, one study found developers spend more than half of their time browsing and reading source code (excluding editing and all other activities in the IDE) (Snipes et al., 2014). This finding supports initiatives to build and promote tools that improve the navigation experience by supporting structural navigation. Other references mentioned in this chapter leveraged usage data to identify opportunities to improve refactoring tools (Vakilian and Johnson, 2014; Murphy-Hill, Jiresal and Murphy, 2012). By identifying task contexts through usage data, Mylyn made improvements to how developers manage their code and task context (Kersten and Murphy, 2006).

If the last two decades could be labeled the era of big data collection, the next two decades will surely be labeled as the era of smarter big data analysis. Many questions still remain: How do we balance data privacy and data richness? What are the long term effects of developer monitoring? How can we maximize the value of data collection for as many questions as possible, and reduce the strain on developers providing the data? How can we provide right data to right person at right time with the least effort? Answering these questions will help the community advance in usage data collection and analysis.

Usage data, while now widely collected, still remains largely an untapped resource by practitioners and researchers. In this chapter, we have explained how to collect and analyze usage data, which we hope will increase the ease with which you can collect and analyze your own usage data.

## 7. Acknowledgements

The authors would like to thank the software developer community's contribution in sharing their usage data and the support of their respective institutions for the research work behind this chapter. This material is based in part upon work supported by the National Science Foundation under grant number 1252995.

## 8. Code Listings

The following code listings show the code for Build It Yourself for Visual Studio in section 3.4.
Listing for CollectorPackage.cs

```
1  using System;
2  using System.Diagnostics;
3  using System.Globalization;
4  using System.Runtime.InteropServices;
5  using System.ComponentModel.Design;
6  using Microsoft.Win32;
7  using Microsoft.VisualStudio;
8  using Microsoft.VisualStudio.Shell.Interop;
9  using Microsoft.VisualStudio.OLE.Interop;
10 using Microsoft.VisualStudio.Shell;
11 using Monitor;
12
13 namespace Microsoft.Collector
14 {
15     /// <summary>
16     /// This is the class that implements the package exposed by this assembly.
17     ///
18     /// The minimum requirement for a class to be considered a valid package for Visual Studio
19     /// is to implement the IVsPackage interface and register itself with the shell.
20     /// This package uses the helper classes defined inside the Managed Package Framework (MPF)
21     /// to do it: it derives from the Package class that provides the implementation of the
22     /// IVsPackage interface and uses the registration attributes defined in the framework to
23     /// register itself and its components with the shell.
24     /// </summary>
25     // This attribute tells the PkgDef creation utility (CreatePkgDef.exe) that this class is
```

```
26      // a package.
27      [PackageRegistration(UseManagedResourcesOnly = true)]
28      // This attribute is used to register the information needed to show this package
29      // in the Help/About dialog of Visual Studio.
30      [InstalledProductRegistration("#110", "#112", "1.0", IconResourceID = 400)]
31      // This attribute is needed to let the shell know that this package exposes some menus.
32      [ProvideMenuResource("Menus.ctmenu", 1)]
33      // This attribute starts the package when Visual Studio starts
34      [ProvideAutoLoad("{ADFC4E64-0397-11D1-9F4E-00A0C911004F}")]
35      [Guid(GuidList.guidCollectorPkgString)]
36      public sealed class CollectorPackage : Package
37      {
38          /// <summary>
39          /// Default constructor of the package.
40          /// Inside this method you can place any initialization code that does not require
41          /// any Visual Studio service because at this point the package object is created but
42          /// not sited yet inside Visual Studio environment. The place to do all the other
43          /// initialization is the Initialize method.
44          /// </summary>
45          public CollectorPackage()
46          {
47              Debug.WriteLine(string.Format(CultureInfo.CurrentCulture, "Entering constructor for: {0}",
                        this.ToString()));
48          }
49
50
51
52          ///////////////////////////////////////////////////////////////////////////
53          // Overridden Package Implementation
54          #region Package Members
55
56          /// <summary>
57          /// Initialization of the package; this method is called right after the package is sited, so this is the place
58          /// where you can put all the initialization code that rely on services provided by VisualStudio.
59          /// </summary>
60          protected override void Initialize()
61          {
62              Debug.WriteLine (string.Format(CultureInfo.CurrentCulture, "Entering Initialize() of: {0}",
                        this.ToString()));
63              base.Initialize();
64
65              // Add our command handlers for menu (commands must exist in the .vsct file)
66              OleMenuCommandService mcs = GetService(typeof(IMenuCommandService)) as OleMenuCommandService;
67              if ( null != mcs )
68              {
69                  // Create the command for the menu item.
70                  CommandID menuCommandID = new CommandID(GuidList.guidCollectorCmdSet,
                            (int)PkgCmdIDList.StopMonitoring);
71                  MenuCommand menuItem = new MenuCommand(MenuItemCallback, menuCommandID );
72                  mcs.AddCommand( menuItem );
73              }
74              DataRecorder.Start();
75          }
76          #endregion
77
78          /// <summary>
79          /// This function is the callback used to execute a command when the a menu item is clicked.
80          /// See the Initialize method to see how the menu item is associated to this function using
81          /// the OleMenuCommandService service and the MenuCommand class.
82          /// </summary>
83          private void MenuItemCallback(object sender, EventArgs e)
84          {
85              DataRecorder.Stop();
86              // Show a Message Box to prove we were here
87              IVsUIShell uiShell = (IVsUIShell)GetService(typeof(SVsUIShell));
88              Guid clsid = Guid.Empty;
```

```
89              int result;
90              Microsoft.VisualStudio.ErrorHandler.ThrowOnFailure(uiShell.ShowMessageBox(
91                              0,
92                              ref clsid,
93                              "Collector",
94                              string.Format(CultureInfo.CurrentCulture, "Collector Stopped"),
95                              string.Empty,
96                              0,
97                              OLEMSGBUTTON.OLEMSGBUTTON_OK,
98                              OLEMSGDEFBUTTON.OLEMSGDEFBUTTON_FIRST,
99                              OLEMSGICON.OLEMSGICON_INFO,
100                             0, // false
101                             out result));
102         }
103
104     }
105 }
```

Listing 1: CollectorPackage

Listing for DataRecorder.cs

```
using System;

namespace Monitor
{
    public static class DataRecorder
    {
        public static void Start()
        {
            logDirectoryPath = System.IO.Path.GetTempPath();
            logFileName = System.IO.Path.Combine(logDirectoryPath, "collector " +
                DateTime.Now.ToString("yyyy-MM-dd HH.mm.ss") + ".log");
            try
            {
                using (System.IO.StreamWriter streamWriter = new System.IO.StreamWriter(
                    new System.IO.FileStream(logFileName, System.IO.FileMode.OpenOrCreate,
                        System.IO.FileAccess.Write, System.IO.FileShare.ReadWrite)
                    ))
                {
                    streamWriter.WriteLine("Collector Started");
                }
            }
            catch (System.IO.IOException ioexception)
            {
                Console.WriteLine("Error creating log file " + ioexception);
            }
            myEvents = new MonitoredEventCollection();
            myEvents.RegisterEventInventoryForEventMonitoring();
        }

        public static void Stop()
        {
            myEvents.DeRegisterEventMonitoringForInventory();
            WriteLog("Collector Stopped");
        }

        public static void WriteLog(string logToWrite)
        {
            try
            {
                using (System.IO.StreamWriter streamWriter = new System.IO.StreamWriter(
                    new System.IO.FileStream(logFileName, System.IO.FileMode.Append, System.IO.FileAccess.Write,
                        System.IO.FileShare.ReadWrite)
                    ))
                {
                    streamWriter.WriteLine(logToWrite);
                }
            }
            catch (System.IO.IOException ioexception)
            {
                Console.WriteLine("Error writing to log file " + ioexception);
            }
        }

        static MonitoredEventCollection myEvents;
        private static string logFileName;
        private static string logDirectoryPath;
    }
}
```

Listing 2: DataRecorder

Listing for AbstractMonitoredEvent.cs

```
using System;
using System.IO;
using System.Text;
using System.Xml.Linq;
using System.Xml.Serialization;

namespace Monitor
{

    [XmlInclude(typeof(MonitoredCommandEvent))]
    [XmlRoot(ElementName = "MonitoredEvent", Namespace = "http://Monitor")]
    public abstract class AbstractMonitoredEvent
    {
        /// <summary>
        /// Default constructor to use in serialization
        /// </summary>
        protected AbstractMonitoredEvent()
        {
        }

        // User friendly event name used for recording in logs
        public String EventName { get; set; }

        // Configured classification for the log
        public String Classification { get; set; }

        // Stores information related to artifacts such as window titles active during the event
        public String ArtifactReference { get; set; }

        public void ToLog()
        {
            DataRecorder.WriteLog(String.Join(",", System.DateTime.UtcNow.ToString("u"), this.EventName,
                this.Classification));
        }

        #region event handler registration and disposal

        public virtual bool RegisterEventForMonitoring(object dte)
        {
            return false;
        }


        public void Dispose()
        {

            this.Dispose(true);

            //GC.SuppressFinalize(this);

        }

        protected virtual void Dispose(bool disposing)
        {
            this.isDisposed = true;
        }

        protected bool isDisposed;

        #endregion
    }
}
```

Listing 3: AbstractMonitoredEvent

Listing for MonitoredCommandEvent.cs

```
using System;
using System.Xml.Linq;
using EnvDTE;
using System.Xml.Serialization;

namespace Monitor
{

    [XmlRoot(ElementName = "MonitoredEvent", Namespace = "http://Monitor")]
    public class MonitoredCommandEvent : AbstractMonitoredEvent
    {

        /// <summary>
        /// DTE object EventID integer distinguishes events with a shared GUID.
        /// </summary>
        public int EventID { get; set; }

        /// <summary>
        /// GUID of the DTE event from Visual Studio
        /// </summary>
        public String Guid { get; set; }

        /// <summary>
        /// Default constructor to use in serilization
        /// </summary>
        public MonitoredCommandEvent()
        {
        }

        /// <summary>
        /// Create an object from the Command class of the DTE
        /// </summary>
        /// <param name="DTECommandObj">Command class of the DTE</param>
        public MonitoredCommandEvent(Command DTECommandObj)
        {
            if (DTECommandObj != null)
            {
                this.EventName = DTECommandObj.Name;
                this.Classification = EventName.Split('.')[0]; //use the first part of event name
                this.Guid = DTECommandObj.Guid;
                this.EventID = DTECommandObj.ID;
            }
            else
            {
                throw new ArgumentNullException("DTECommandObj");
            }
        }

        #region Event registration, disposal, and hander
        ///<summary>
        ///The event type object holds the event class type for this interceptor for example CommandEvents
        ///the RegisterEvent method registers the event
        ///</summary>
        private CommandEvents eventTypeObject;

        public override bool RegisterEventForMonitoring(object dte)
        {
            if (!isDisposed && eventTypeObject == null && dte != null)
            {
                eventTypeObject = (dte as DTE).Events.get_CommandEvents(Guid, EventID) as CommandEvents;
            }
            if (eventTypeObject != null)
            {
                eventTypeObject.AfterExecute += new
```

```
                    _dispCommandEvents_AfterExecuteEventHandler(OnAfterExecute);
            }
            return (eventTypeObject != null);
        }


        /// <summary>
        /// Remove the event from the handler list
        /// </summary>
        /// <param name="disposing"></param>
        protected override void Dispose(bool disposing)
        {
            if (eventTypeObject != null)
                eventTypeObject.AfterExecute -= OnAfterExecute;
            this.isDisposed = true;
        }


        /// <summary>
        /// Method receives event after the command completes execution. Adds the end of
        /// the command event to the log
        /// </summary>
        /// <param name="Guid">Guid of the command</param>
        /// <param name="ID">numeric id of the command</param>
        /// <param name="CustomIn"></param>
        /// <param name="CustomOut"></param>
        private void OnAfterExecute(string Guid, int ID, object CustomIn, object CustomOut)
        {
            this.ToLog();
        }
        #endregion
    }
}
```

Listing 4: MonitoredCommandEvent

Listing for MonitoredEventFactory.cs

```
using System;
using System.IO;
using System.Text;
using System.Xml.Linq;
using System.Xml.Serialization;
using EnvDTE;

namespace Monitor
{
    public static class MonitoredEventFactory
    {

        public static AbstractMonitoredEvent GetMonitoredEvent(Command DTECommandObj)
        {
                object eventObj = new MonitoredCommandEvent(DTECommandObj);
                return (AbstractMonitoredEvent)eventObj;

        }

    }
}
```

Listing 5: MonitoredEventFactory

Listing for MonitoredEventCollection.cs

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Reflection;
using System.Xml;
using System.Xml.Linq;
using EnvDTE;
using Microsoft.VisualStudio.Shell;
using System.Xml.Serialization; //12.0

namespace Monitor
{
    public class MonitoredEventCollection
    {
        /// <summary>
        /// Object to store all the MonitoredEvents we have on file
        /// </summary>
        private List<AbstractMonitoredEvent> IDEEventListenerRegistry;

        /// <summary>
        /// Constructor that reads events from a file or queries Visual Studio for the command events
        /// if the file does not exist. Then saves the events to the file for next time.
        /// </summary>
        public MonitoredEventCollection()
        {
            String EventInventoryFilePath =
                Path.Combine(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location),
                "CommandGUIDs.xml");
            MonitoredEventCollectionInitialize(EventInventoryFilePath);
        }

        private void MonitoredEventCollectionInitialize(String EventInventoryFilePath) {
            if (File.Exists(EventInventoryFilePath)) {
                IDEEventListenerRegistry = LoadEventsFromFile(EventInventoryFilePath);
            }
            else {
                IDEEventListenerRegistry = QueryVSForAllDTECommands();

            }
            if (IDEEventListenerRegistry != null) {
                saveEventInterestTable(IDEEventListenerRegistry, EventInventoryFilePath);
            }
        }

        private List<AbstractMonitoredEvent> LoadEventsFromFile(string filepath)
        {
            try
            {
                List<AbstractMonitoredEvent> eventList = new List<AbstractMonitoredEvent>();
                var serializer = new System.Xml.Serialization.XmlSerializer(typeof(List<AbstractMonitoredEvent>));
                using (Stream file = new FileStream(filepath, FileMode.Open, FileAccess.Read))
                {
                    eventList = (List<AbstractMonitoredEvent>)serializer.Deserialize(file);
                }
                return eventList;
            }
            catch (System.IO.IOException)
            {
                Console.WriteLine("Error opening file with event inventory" + filepath);
                return null;
            }
        }

        private void saveEventInterestTable(List<AbstractMonitoredEvent> eventList, string filepath)
```

```
63          {
64              try
65              {
66                  var serializer = new System.Xml.Serialization.XmlSerializer(typeof(List<AbstractMonitoredEvent>));
67                  using (Stream file = new FileStream(filepath, FileMode.Create, FileAccess.Write))
68                  {
69                      serializer.Serialize(file, eventList);
70                      file.Flush();
71                  }
72              }
73              catch (System.IO.IOException)
74              {
75                  Console.WriteLine("Error creating file for storing monitored events with file path:" + filepath);
76              }
77
78          }
79          /// <summary>
80          /// Query the DTE Commands object for all events it provides. Could be useful to determine whether new
                commands from
81          /// Add-Ins or Extensions appeared since we built the inventory. Returns a collection of Events with
                Immediate type
82          /// </summary>
83          /// <returns>List of AbstractMonitoredEvents in the DTE object</returns>
84          private List<AbstractMonitoredEvent> QueryVSForAllDTECommands()
85          {
86              List<AbstractMonitoredEvent> EventList = new List<AbstractMonitoredEvent>();
87              DTE dteobj = tryGetDTEObject();
88              if (dteobj != null)
89              {
90
91                  try
92                  {
93                      foreach (Command DTE_CommandEventObj in dteobj.Commands)
94                      {
95                          AbstractMonitoredEvent NewEvent =
                                MonitoredEventFactory.GetMonitoredEvent(DTE_CommandEventObj);
96                          if (NewEvent != null)
97                          {
98                              EventList.Add(NewEvent);
99                          }
100                     }
101                 }
102                 //This exception happens during dispose/finalize when VS exits, just return null
103                 catch (System.Runtime.InteropServices.InvalidComObjectException)
104                 {
105                     return null;
106                 }
107             }
108             return EventList;
109         }
110
111
112         /// <summary>
113         /// Gets a DTE object for the currently running Visual Studio instance. Requires references
114         /// to EnvDTE, Microsoft.VisualStudio.Shell.12.0, and Microsoft.VisualStudio.OLE.Interop.
115         /// </summary>
116         /// <returns></returns>
117     private static DTE tryGetDTEObject()
118     {
119         DTE dteobj=null;
120             try
121             {
122                 dteobj =
                        ((EnvDTE.DTE)ServiceProvider.GlobalProvider.GetService(typeof(EnvDTE.DTE).GUID)).DTE;
123             }
124             catch (NullReferenceException)
```

```
125             { }
126             catch (System.Runtime.InteropServices.InvalidComObjectException)
127             { }
128             catch (System.Runtime.InteropServices.COMException)
129             { }
130          return dteobj;
131       }
132
133         public bool RegisterEventInventoryForEventMonitoring()
134         {
135
136             DTE dteobj = tryGetDTEObject();
137             bool somethingRegistered = false;
138             if (dteobj != null && IDEEventListenerRegistry != null && IDEEventListenerRegistry.Count > 0)
139             {
140
141                 foreach (AbstractMonitoredEvent command in IDEEventListenerRegistry)
142                 {
143                     if (command.RegisterEventForMonitoring(dteobj))
144                     {
145                         somethingRegistered = true;
146                     }
147                 }
148
149
150             }
151
152             return somethingRegistered;
153         }
154
155         public void DeRegisterEventMonitoringForInventory()
156         {
157
158             foreach (AbstractMonitoredEvent monitoredEvent in IDEEventListenerRegistry)
159             {
160                 monitoredEvent.Dispose();
161             }
162
163         }
164
165     }
166 }
```

Listing 6: MonitoredEventCollection

## References


Basili, V. R. and Rombach, H. D. (1988), 'The TAME project: towards improvement-oriented software environments', *Software Engineering, IEEE Transactions on* **14**(6), 758–773.

Biehl, J. T., Czerwinski, M., Smith, G. and Robertson, G. G. (2007), Fastdash: A visual dashboard for fostering awareness in software teams, *in* 'Proceedings of the SIGCHI Conference on Human Factors in Computing Systems', CHI '07, ACM, New York, NY, USA, pp. 1313–1322.

Carter, J. and Dewan, P. (2010), Are you having difficulty?, *in* 'Proceedings of the 2010 ACM conference on Computer supported cooperative work', CSCW '10, ACM, New York, NY, USA, pp. 211–214.

Coman, I. D. and Sillitti, A. (2008), Automated identification of tasks in development sessions, *in* 'Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension', ICPC '08, IEEE Computer Society, Washington, DC, USA, pp. 212–217.

del Galdo, E. M., Williges, R. C., Williges, B. H. and Wixon, D. R. (1986), An Evaluation of Critical Incidents for Software Documentation Design, *in* 'Proc. HFES', pp. 19–23.

Flanagan, J. C. (1954), 'The Critical Incident Technique', *Psychological Bulletin* pp. 327–358.

Foster, S., Griswold, W. G. and Lerner, S. (2012), Witchdoctor: Ide support for real-time auto-completion of refactorings, *in* 'Software Engineering (ICSE), 2012 34th International Conference on', pp. 222–232.

Fritz, T., Ou, J., Murphy, G. C. and Murphy-Hill, E. (2010), A degree-of-knowledge model to capture source code familiarity, *in* 'Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1', ICSE '10, ACM, New York, NY, USA, pp. 385–394.

Ge, X., DuBose, Q. and Murphy-Hill, E. (2012), Reconciling manual and automatic refactoring, *in* 'Software Engineering (ICSE), 2012 34th International Conference on', pp. 211–221.

Johnson, P. M., Kou, H., Agustin, J., Chan, C., Moore, C., Miglani, J., Zhen, S. and Doane, W. E. J. (2003), Beyond the personal software process: Metrics collection and analysis for the differently disciplined, *in* 'Proceedings of the 25th international Conference on Software Engineering', IEEE Computer Society.

Kersten, M. (2007), Focusing knowledge work with task context, PhD thesis, The University of British Columbia.

Kersten, M. and Murphy, G. C. (2005), Mylar: A degree-of-interest model for ides, *in* 'Proceedings of the 4th International Conference on Aspect-oriented Software Development', AOSD '05, ACM, New York, NY, USA, pp. 159–168.

Kersten, M. and Murphy, G. C. (2006), Using task context to improve programmer productivity, *in* 'Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering', SIGSOFT '06/FSE-14, ACM, New York, NY, USA, pp. 1–11.

Kou, H., Johnson, P. and Erdogmus, H. (2010), 'Operational definition and automated inference of test-driven development with Zorro', *Automated Software Engineering* **17**(1), 57–85.

Lee, Y. Y., Chen, N. and Johnson, R. E. (2013), Drag-and-drop refactoring: Intuitive and efficient program transformation, *in* 'Software Engineering (ICSE), 2013 35th International Conference on', pp. 23–32.

Liu, H., Gao, Y. and Niu, Z. (2012), An initial study on refactoring tactics, *in* 'Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual', IEEE, pp. 213–218.

Maalej, W., Fritz, T. and Robbes, R. (2014), *Collecting and processing interaction data for recommendation systems*, Springer, chapter Recommendation Systems in Software Engineering.

Murphy, G. C., Kersten, M. and Findlater, L. (2006), 'How are Java software developers using the Elipse IDE?', *IEEE Software* **23**(4), 76–83.

Murphy, G. C., Viriyakattiyaporn, P. and Shepherd, D. (2009), Using activity traces to characterize programming behaviour beyond the lab, *in* 'Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on', IEEE, pp. 90–94.

Murphy-Hill, E. and Black, A. P. (2008), Breaking the barriers to successful refactoring: Observations and tools for extract method, *in* 'Proceedings of the 30th International Conference on Software Engineering', ICSE '08, ACM, New York, NY, USA, pp. 421–430.

Murphy-Hill, E., Jiresal, R. and Murphy, G. C. (2012), Improving Software Developers' Fluency by Recommending Development Environment Commands, *in* 'Foundations of Software Engineering'.

Murphy-Hill, E., Parnin, C. and Black, A. P. (2012*a*), 'How We Refactor, and How We Know It', *IEEE Transactions on Software Engineering* **38**, 5–18.

Murphy-Hill, E., Parnin, C. and Black, A. P. (2012*b*), 'How we refactor, and how we know it', *Software Engineering, IEEE Transactions on* **38**(1), 5–18.

Murphy-Hill, E. R., Ayazifar, M. and Black, A. P. (2011), Restructuring software with gestures, *in* 'VL/HCC', pp. 165–172.

Negara, S., Chen, N., Vakilian, M., Johnson, R. E. and Dig, D. (2013), A Comparative Study of Manual and Automated Refactorings, *in* 'Proceedings of the European Conference on Object-Oriented Programming (ECOOP)', pp. 552–576.

Negara, S., Vakilian, M., Chen, N., Johnson, R. E. and Dig, D. (2012), Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?, *in* 'Proceedings of the European Conference on Object-Oriented Programming (ECOOP)', pp. 79–103.

Parnin, C. and Rugaber, S. (2011), 'Resumption strategies for interrupted programming tasks', *Software Quality Journal* **19**(1), 5–34.

Robillard, M. P., Coelho, W. and Murphy, G. C. (2004), 'How effective developers investigate source code: an exploratory study', *IEEE Trans. Softw. Eng.* **30**, 889–903.

Shattuck, L. G. and Woods, D. D. (1994), The Critical Incident Technique: 40 Years Later, *in* 'Proc. HFES', pp. 1080–1084.

Snipes, W., Nair, A. and Murphy-Hill, E. (2014), Experiences Gamifying Developer Adoption of Practices and Tools, *in* 'Software Engineering, 2014. ICSE 2014. IEEE 36st International Conference on'.

Vakilian, M., Chen, N., Moghaddam, R. Z., Negara, S. and Johnson, R. E. (2013), A Compositional Paradigm of Automating Refactorings, *in* 'Proceedings of the European Conference on Object-Oriented Programming (ECOOP)', pp. 527–551.

Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P. and Johnson, R. E. (2012), Use, Disuse, and Misuse of Automated Refactorings, *in* 'Proceedings of the International Conference on Software Engineering (ICSE)', pp. 233–243.

Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Zilouchian Moghaddam, R. and Johnson, R. E. (2011), The Need for Richer Refactoring Usage Data, *in* 'Proceedings of the Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)', pp. 31–38.

Vakilian, M. and Johnson, R. E. (2014), Alternate refactoring paths reveal usability problems, *in* 'Proceedings of the International Conference on Software Engineering (ICSE) (To Appear)', pp. 1–11.

Ying, A. T. and Robillard, M. P. (2011), The influence of the task on programmer behaviour, *in* 'Program Comprehension (ICPC), 2011 IEEE 19th International Conference on', IEEE, pp. 31–40.

Yoon, Y. and Myers, B. A. (2011), Capturing and Analyzing Low-level Events from the Code Editor, *in* 'Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools', pp. 25–30.

Yoon, Y. and Myers, B. A. (2012), An Exploratory Study of Backtracking Strategies Used by Developers, *in* 'Proceedings of the 5th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)', pp. 138–144.

Yoon, Y., Myers, B. A. and Koo, S. (2013), Visualization of Fine-Grained Code Change History, *in* 'Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)', pp. 119–126.

Zou, L. and Godfrey, M. W. (2012), An industrial case study of coman's automated task detection algorithm: What worked, what didn't, and why, *in* 'Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)', ICSM '12, IEEE Computer Society, Washington, DC, USA, pp. 6–14.