A Newbie's Guide to Eclipse APIs

Reid Holmes and Robert J. Walker
Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
rtholmes,rwalker@cpsc.ucalgary.ca

ABSTRACT

Eclipse has evolved from a fledgling Java IDE into a mature software ecosystem. One of the greatest benefits Eclipse provides developers is flexibility; however, this is not without cost. New Eclipse developers often find the framework to be large and confusing. Determining which parts of the framework they should be using can be a difficult task as Eclipse documentation tends to be either very high-level, focusing on the design of the framework, or low-level, focusing on specific APIs. We have developed a tool called PopCon that provides a bridge between high-level design documentation and low-level API documentation by statically analyzing a framework and several of its clients and providing a ranked list of the relative popularity of its APIs. We have applied PopCon to the Eclipse framework for this challenge to help newbie Eclipse developers identify some of the most relevant APIs for their tasks.

Categories and Subject Descriptors

H.5.2 [**User Interfaces**]: Training, help, and documentation

General Terms

Documentation

Keywords

API popularity, PopCon, mining software repositories

1. INTRODUCTION

Eclipse has evolved into a very large framework. Europa, its most recent release, contained over 1,200 individual plug-ins. Eclipse is a flexible system that provides many great features developers can leverage to quickly make impactful programs; however, Eclipse's flexibility and breadth of functionality makes it difficult for new developers to understand. To combat this problem Eclipse provides a large amount of documentation. This tends to be at two levels: (1) high-level design documentation that outlines the major components of the Eclipse system and (2) low-level API documentation that specifies what each API does. These two types of documentation result in a conceptual gap that the developer must overcome: the high-level documentation suggests particular plug-ins they should focus on but looking at the API documentation for that plug-in they find hundreds of types and methods, alphabetically sorted, with no indication of which they should investigate first. This problem is particularly evident on the Javadoc package-overview page as it lists all of the interfaces and classes declared within the package giving only short (typically less than 10 word) descriptions of each. Our tool, called PopCon, has been developed to address this shortcoming: by enumerating the API usage information in its database, PopCon can provide the developer with an overview of which APIs are most used for any particular package, class, or interface. In this way PopCon can provide a more meaningful ranking of the individual APIs helping a developer focus their investigation on those API elements that are most often used without being overwhelmed by the less important elements. PopCon can also provide the source code for each usage, that the developer can leverage as an implicit example of how the API can be used.

The Eclipse framework is particularly well-suited for this type of analysis as it is comprised of a small kernel of core code that is extended by many different plug-ins. This means that, by analyzing how Eclipse developers make use of the framework itself, it is possible to get a comprehensive understanding of the most important APIs. With the advent of the Eclipse release train (first Callisto, then Europa, and now Ganymede) the framework also ships with a huge amount of extra code (e.g., CDT, WST, TPTP, and RSE) that PopCon can analyze to further refine its results.

For the Mining Software Repositories 2008 General Challenge, we have used PopCon to analyze Eclipse Europa to infer the most used APIs for several major plug-ins. These APIs represent often-used points that a newbie developer can start from when investigating a new development task where they have only a high-level understanding of what they want to do. This ordering gives them insight into the relative importance of an API and relieves them from either manually searching for examples or investigating each API one-at-a-time.

PopCon has been designed to help a developer, who is trying to perform a particular task, gain insight into which parts of a large framework are the most-often used. As these tasks tend to be specific, PopCon has not been designed to infer generally 'surprising' results; in contrast, it provides specific insights that may help guide a developer in their current task. The data we have provided for this challenge should be considered in this way: *If I were trying to figure out how to use this package, class, or interface would this information be useful to me?*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MSR'08, May 10-11, 2008, Leipzig, Germany.
Copyright 2008 ACM 978-1-60558-024-1/08/05 ...$5.00.

2. POPCON

PopCon is a client-server system: the server contains many structural facts extracted from software systems and the web-based client allows the developer to interact with aggregate usage information in an online fashion. Besides the basic structural extraction, PopCon does not pre-compute any of its results enabling additional systems to be added to the database at anytime. An in-depth discussion of how PopCon works has been previously reported [2].

2.1 Extracting Structure

PopCon uses the lightweight static analysis engine created for the Strathcona Example Recommendation System [1]. The structure crawler extracts information about various program entities (packages, classes, interfaces, fields and methods) and the interactions between them (contains, inheritance, references, calls, and overrides). An Eclipse plug-in has been developed that will extract this structural information, and its associated source code, automatically using the standard Eclipse export functionality. This information is stored in a compressed file that can be automatically uploaded to the server. A PopCon server can support a whole team; there is no need for individual developers to run separate servers or interact with its administrative component, unless they want to add new source code. The structure for Eclipse Europa, which contains several million lines of source code, can be extracted and uploaded to a PopCon database in approximately 4 hours using one inexpensive machine; the developer need only be involved in this process for two five-minute periods (initiating the extraction and the database population).

2.2 Displaying Results

Generally developers start interacting with PopCon by asking for a summary for a particular package. At this level PopCon provides an overview of the most used APIs within that package. This overview includes graphs representing the most extended or implemented types, most overridden methods, most called methods, and most referenced fields in the package. An example of one of these overview graphs is given in Figure 1. In this case the graph represents the most-called methods in the `swt.browser` package. By clicking on any mentioned class or interface the developer is then taken to an overview page for the selected entity. This page describes the relative popularity of any of the methods or fields declared by the entity. By design, PopCon only shows a limited amount of data to the developer in an effort to avoid adding to the information overload they already face.

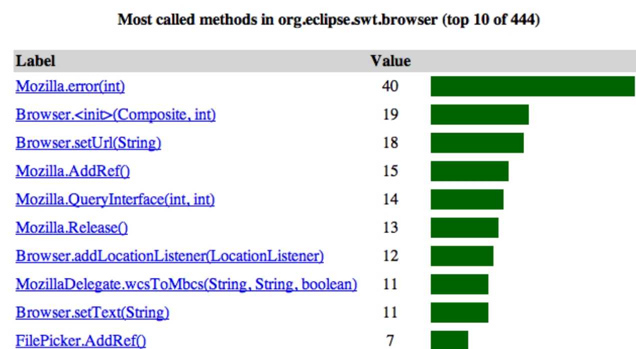

Figure 1: Most-called methods in `swt.browser`

3. ECLIPSE NEWBIE API GUIDE

For this challenge we chose to focus on providing usage overviews for several major Eclipse plug-ins. While PopCon by default shows "Top 10" lists, we have further reduced many of these to "Top 5" lists to save space. Using the basic Eclipse help functionality developers can easily discover each of these plug-ins and what high-level functionality they provide. PopCon can then help the developer identify which low-level APIs they should focus their investigation on.

3.1 Java Development Tools (JDT)

The non-internal API for JDT consists of 41 packages, 205 interfaces, 631 classes, 8897 methods, and 4668 fields. Scanning the overview page for the `org.eclipse.jdt` package, the developer can quickly focus on some of the most important classes and interfaces in this package (and its sub-packages). `ASTVisitor`, `Expression`, `Statement`, `ASTNode`, and `IJavaElement` are all significant JDT classes and PopCon validates this fact (Table 1).

Element	Count
ASTVisitor	100
SelectionDispatchAction	74
Expression	26
SearchRequestor	25
IElementChangeListener	24
Statement	22
ASTNode	19
IClassFileAttribute	16
IJavaElement	13
StandardJavaElementContentProvider	13

Table 1: Most-extended/-implemented JDT classes/interfaces.

The most-called methods in the package (Table 2) give a glimpse into how these types are being used by existing systems. The first two are of particular importance: when working with `ASTNode`s, the `accept(ASTVisitor)` method is key to working with any of `ASTNode`'s child types. `IJavaElement.getJavaProject()` is another very useful library method that is often used but can be easily lost in the other 18 methods defined by this interface.

Element	Count
IJavaElement.getJavaProject()	526
ASTNode.accept(ASTVisitor)	513
ASTNode.getStartPosition()	434
IJavaElement.getElementName()	398
IJavaElement.exists()	384

Table 2: Most-called JDT methods.

If the developer wants to see how the `ASTVisitor` is being used, he can look at the summary page for the class (Table 3). This abstract class can be daunting because it contains 171 methods; PopCon is able to quickly suggest that some methods are used much more often than others. While a developer may not be surprised by the importance of `visit(TypeDeclaration)` or `visit(MethodDeclaration)`, he may have overlooked the importance of `visit(SimpleName)`—the use of which greatly simplifies many basic AST analyses.

Element	Count
ASTVisitor.visit(SimpleName)	60
ASTVisitor.visit(MethodInvocation)	47
ASTVisitor.visit(TypeDeclaration)	41
ASTVisitor.visit(MethodDeclaration)	36
ASTVisitor.visit(AnonymousClassDeclaration)	32

Table 3: Most-overridden ASTVisitor methods.

3.2 Standard Widget Toolkit (SWT)

The non-internal API for SWT consists of 13 packages, 47 interfaces, 401 classes, 5666 methods, and 2533 fields. Looking at the most-extended/-implemented classes and interfaces (Table 4), the developer can quickly see that SWT makes extensive use of listeners. From this list the developer can also infer that the basic widget types are usually used through calling, not through inheritance.

Element	Count
SelectionAdapter	1259
Listener	458
ModifyListener	365
SelectionListener	303
DisposeListener	239

Table 4: Most-extended/implemented SWT classes/interfaces.

Interestingly, the most-called methods in SWT (Table 5) do not relate to the listeners at all but instead pertain to widgets and how they are laid out. Heavy usage of the `GridData` class indicates that `GridLayout` is the most-used layout system for SWT-based applications. Furthermore, a widget's layout can be modified by applying `GridData` objects to them via the `setLayoutData` method.

Element	Count
Control.setLayoutData(Object)	2616
GridData.<init>(int)	1851
Composite.setLayout(Layout)	1749
Composite.<init>(Composite,int)	1410
Widget.getDisplay()	1401

Table 5: Most-called SWT methods.

3.3 JFace

The non-internal API for the JFace plug-ins consists of 39 packages, 279 interfaces, 1035 classes, 10671 methods, and 2901 fields. The most-inherited elements list provides a cross-section of many interesting elements in JFace (Table 6). These include the importance of the `Action` infrastructure, how threads are managed by the UI (`RunnableWithProgress`, `SafeRunnable`), how JFace notifies various widgets of state changes (`ISelectionChanged-Listener`, `IPropertyChangeListener`), and how the document provider infrastructure works (`LabelProvider`, `IStructured-ContentProvider`, `ITreeContentProvider`).

JFace uses constants heavily to convey specific messages between its various components; the developer can get a quick overview of the most-used fields by scanning through the most-referenced field list (Table 7).

Element	Count
Action	912
ISelectionChangedListener	404
IRunnableWithProgress	308
IPropertyChangeListener	289
LabelProvider	269
IStructuredContentProvider	174
ViewerFilter	152
IMenuListener	142
ITreeContentProvider	138
SafeRunnable	121

Table 6: Most-extended/-implemented JFace classes/interfaces.

Element	Count
Window.OK	529
IDialogConstants.OK_ID	251
Position.offset	156
StructuredSelection.EMPTY	143
IDialogConstants.CANCEL_LABEL	140

Table 7: Most-referenced JFace fields.

3.4 User Interface (UI)

The non-internal API for the UI plug-ins consists of 67 packages, 353 interfaces, 1435 classes, 12261 methods, and 4530 fields. The most-called UI methods (Table 8) gives insight into how to get handles to the Eclipse workbench, preferences store, and help system.

Element	Count
PlatformUI.getWorkbench()	2181
IWorkbench.getHelpSystem()	1105
AbstractUIPlugin.getPreferenceStore()	790
IWorkbenchHelpSystem.setHelp(Control, String)	636
WorkbenchPart.getSite()	478

Table 8: Most-called UI methods.

By looking at the inheritance overview (Table 9), the developer's curiosity in the `AbstractUIPlugin` class is piqued and he decides to investigate it further.

Element	Count
IWorkbenchPreferencePage	121
AbstractUIPlugin	116
ActionFactory	81
WorkspaceModifyOperation	80
IWorkbenchWindowActionDelegate	69

Table 9: Most-extended/-implemented UI classes/interfaces.

Examining the most-overridden methods for `AbstractUIPlugin` (Table 10), the developer can see that the first three methods (the constructor, `start(...) and stop(...)`) are used far more often than the the other 8 methods.

Element	Count
AbstractUIPlugin.<init>()	130
AbstractUIPlugin.stop(BundleContext)	102
AbstractUIPlugin.start(BundleContext)	101
AbstractUIPlugin.createImageRegistry()	8
AbstractUIPlugin.initializeImageRegistry(ImageRegistry)	3

Table 10: Most-overridden `AbstractUIPlugin` methods.

4. DISCUSSION

PopCon can be confounded in two particular cases. First, it cannot effectively surface important new APIs. This is because older APIs tend to be more heavily used than newer ones, even if the newer ones are the correct ones to use. Secondly, PopCon can be confused by frequently-used, but ultimately unimportant methods such as `Widget.isDisposed()` and `Widget.dispose()`.

To an expert's eye, PopCon's results may not seem particularly surprising. But to a novice's eye, such results could well be the key to tackling a difficult task. That such a simple approach to mining a software repository can yield powerful results is important: complex visualizations and analyses were not needed—and likely not appropriate since the specific tasks to be tackled would not be known a priori. The results of PopCon could well be used to inform the process of documenting the system for a newbie.

Nevertheless, PopCon will be extended to allow the filtration of its results on a temporal basis. Results will be compiled that involve only a particular release or set of releases of an API, for example, or that have been added to the repository within a particular timeframe. Such information is currently maintained within the repository, but PopCon does not leverage it at present.

PopCon can also be used by API owners to get a sense for how frequently their APIs are being used. They can also navigate to the source code for each usage to see how other developers are using their APIs and see if they are using them correctly. This information can be valuable for the developer as it is often difficult to manually collect and enumerate this data manually.

5. CONCLUSION

We have applied PopCon to this challenge to help developers bridge the gap between high-level and low-level documentation by mining the structural elements in the Eclipse source code. By enumerating the usage of various APIs, PopCon is able to give the developer an indication of the relative importance of various APIs for a particular plug-in, package, class, or interface. This knowledge can help them prioritize their investigation efforts when they are working with unfamiliar portions of Eclipse. An overview of the most popular APIs for various major Eclipse components have been provided to provide insight into the quality of the information that PopCon returns.

6. REFERENCES

[1] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[2] Reid Holmes and Robert J. Walker. Informing Eclipse API production and consumption. In *Proceedings of the 5th OOPSLA Workshop on Eclipse Technology eXchange*, pages 70–74, 2007.