Informing Eclipse API Production and Consumption

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

Application programming interfaces (APIs) inform application developers as to the functionality provided by a library and how to interact with it. APIs are a double-edged sword: if they do not permit the needed functionality to be accessed and adapted as needed, they are obstructing; if they permit all things to all people, they are complex, leading application developers to have difficulty understanding how to use them correctly. Thus, the developers of APIs have a delicate balance to strike between providing configurable functionality and simple interfaces. Inevitably, the wrong balance is sometimes chosen, as the actual usage is different from the expected usage; APIs need to evolve, or to be re-documented to account for this disparity. In this paper we propose a simple technique for quantitatively determining how existing APIs are used, and demonstrate its application to Eclipse. This technique would enable application developers to more easily understand how others have used the APIs and would allow API developers to more easily understand how their APIs are being used.

1. INTRODUCTION

Application programming interfaces (APIs) inform application developers as to the functionality provided by a library/framework and how to use or extend it [12]. APIs are a double-edged sword: if they do not permit functionality to be accessed and adapted as needed, they are obstructing [3]; if they support multiple, tailorable modes of interaction, it is harder for application developers to understand how to use them correctly [1, 10]. Thus, API developers have a delicate balance to strike between providing configurable functionality and simple interfaces [6]. Inevitably, the wrong balance is sometimes chosen, as the actual usage is different from the expected usage; APIs need to evolve or to be re-documented to rectify this disparity [21, 2, 4]. Finding the right balance is difficult for API producers because they cannot directly see how their APIs are being used; API consumers can also benefit from seeing how other developers have used the APIs they are interested in.

Even assuming that one has access to (other) developers' source code, the difficulty of these tasks can be compounded by information overload: checking each project against each API of interest manually is time-consuming and error-prone. API consumers would like to know which APIs past developers have found most useful; API producers would like to know which of their APIs have been most useful to their clients. For example, a developer creating an Eclipse plug-in to render a web page might want to know which types in `org.eclipse.swt.browser` are most often used by existing plug-ins. At the same time, the owner of these APIs would like to know which of their classes and methods are being most used by their clients. Manually extracting this information from a corpus of source code is impracticably laborious.

Existing approaches tend to do one of three things: reverse engineer entire frameworks, find frequent patterns, or locate specific examples. Reverse engineering approaches attempt to infer design- and/or requirements-level information without reference to actual usages of APIs (e.g., [16]) or without abstracting away the least important parts to the developer (e.g., [15]). Frequent pattern mining has been used to determine common, simple patterns of usage [14, 22] but lacks the ability either to be tailored to the user's specific interests (results are pre-computed) or to locate unusual situations. Example browsing based on lexical searches[1] can easily be confounded by apparent name collisions that are resolvable only when a programming language's semantics are considered. Our previous work [8] considers how task-relevant examples can be found using a developer's simple code skeleton; this does not help to answer the more general question of how others are using an API.

The frequency of API use ("popularity") can help API developers prioritize their bug-fixing efforts by repairing frequently-used APIs over "less important" APIs. It can also help application developers to focus their investigative efforts on APIs that more developers have found useful in the past, rather than wade through large API descriptions to find what they need.

To determine API popularity, we use static structure as the basis of our analysis. After extracting the structure from a variety of projects that utilize the same framework, we construct a repository that can be queried to determine how these different projects make use of the APIs provided by the framework. We aggregate the results of these usages to see how heavily an API is used. API consumers can use this popularity measure to focus their investigative efforts when trying to identify the "important" APIs within a package or project. An API producer can use this information to see which of their APIs are being most used. Both groups can also view the specific API usage to either learn how to use the API (consumer) or see how consumers have been using, or potentially misusing, the API (producer). A web interface that allows users to both query and browse the API popularity measures is provided, enabling users to either make targeted queries or more broadly investigate modules of interest. Our approach allows the user to spec-

[1]For example, Krugle.com, Koders.com, or Google Code Search.

OOPSLA'07 Eclipse Technology Exchange (eTX),
October 21, 2007, Montréal, Canada.

ify an API of interest; to generate a bar chart that rank-orders the types, methods, and/or fields in that API based on their popularity; and to refine this view through compound queries based on a variety of properties that the user is interested in.

The Eclipse framework is a perfect candidate for this kind of investigation as it provides a rich set of APIs that can be used for a variety of tasks. As Eclipse has been widely adopted, it is easy to find source code that leverages its APIs, including the Eclipse implementation itself.

This paper outlines our approach and an initial implementation of the tool. Section 2 outlines a motivational scenario in which our approach could be applied. Section 3 describes the facts and measures used in this approach. Section 4 describes four different tasks for which this approach would be suitable. Section 5 describes other work similar to this project. Open issues and future work are discussed in Section 6.

This paper contributes a lightweight approach to informing both API developers and application developers of how others are using an API.

2. MOTIVATIONAL SCENARIO

In this section, we provide a motivational scenario from the point of view of two developers, one an API consumer and the other an API producer.

The Eclipse plug-in developer (the API consumer) is interested in adding a control to their plug-in that is capable of displaying web pages. By looking at the JavaDoc for the Standard Widget Toolkit (SWT), the API consumer quickly notices the `org.eclipse.swt.browser` package that sounds promising for the desired functionality. As the package contains 7 interfaces and 9 classes with a total of 100 methods, it is difficult for the developer to decide how to best spend his time investigating the API. The primary obstacle for the API consumer is abundance of information: between documentation, newsgroups, and the framework source code itself, it is difficult for him to prioritize his investigative efforts to avoid "analysis paralysis".

Conversely, the owner of the Eclipse `org.eclipse.swt.browser` API would like to know how her API is being used in production plug-ins. Specifically, she would like to know if some parts of the API are more used than others, and if some parts are not used at all. She wants this usage information so she can prioritize bug fixes that clients would most benefit from. Her only avenue for getting this kind of information is to check the Eclipse Bugzilla and newsgroups; if she is really interested, she can also post a message to the SWT newsgroup and hope for a response. The primary obstacle for the API producer is the lack of information. It is difficult for her to get a clear indication of how her API is being used as she simply created the API and shipped it; no formal feedback loop exists to keep her informed as to how her API is being used in the wild.

Our approach for generating this information for these developers is detailed in Section 3. We return to this scenario in Section 4 to see how this information can help in these cases.

3. THE POPCON TOOL

Our approach aims to provide developers with quantitative data about various APIs' frequency of use, which we refer to as the APIs' popularity. Our conjecture is that APIs that are used more frequently are relatively more important (from the perspective of writing code) than those that are used less frequently. Before computing the popularity of an API, its structure must first be extracted (Section 3.1). The measurement of popularity is described in Section 3.2. The developer interacts with the tool by browsing for general information or querying for specific information (Section 3.3). The results of their investigations are presented as a series of charts (Section 3.4).

3.1 Fact extraction

Structural relationships form the basis of all of the attributes considered in our computation of API popularity. These structural relationships are extracted statically from the source code; as such, dynamic information is not considered. We consider four primary relationships: inheritance, including class extension and interface implementation; overriding, including implementing a method from an interface or overriding a method from a superclass; calling from one method to another method; and referencing fields. During the static analysis phase we extract these relationships from the source code along with the identity of each project, its package structure and the names of all its classes (and their declared methods and fields). The data model we use is built upon that previously developed and extensively tested for the Strathcona tool [8]. The structural data is extracted from source code using an Eclipse AST Visitor. The Strathcona analyzer has been augmented to include overrides-relationship information and to keep track of class and method modifiers (public, private, protected, static, and abstract).

These relationships and project identities comprise the facts that can be considered in measuring popularity. These facts are stored in a database to enable them to be queried interactively. As the database is frequently queried but infrequently changed, it is optimized for read access through the heavy usage of indices. In addition to the structural facts, each source file is also stored in the file system so it can be retrieved if required.

3.2 Computing API popularity

We calculate API popularity in a straightforward way. For each structural element in the system, we count how many times it has been the target of one of the structural relationships described in Section 3.1. For each class we count how many times it has been extended by a subclass; for each interface we count how many times it has been implemented. For each method we count how many times it has been called, as well as how many times it has been overridden by a method in a subclass. For each field we count how many times it has been referenced.

These popularity values are computed in an online manner; this permits the user to restrict both which parts of the data set they are interested in seeing (e.g., the most overridden method in `org.eclipse.jdt.core`) and which pieces of data contribute to the popularity count itself (e.g., ignore any method calls from `internal` packages). Online computation of the popularity measure supports rich querying and browsing of the data (Section 3.3).

The popularity measure for a class or interface enumerates the number of times it has been extended or implemented. The popularity measure for a field enumerates the number of times that field has been referenced. A method has two popularity measures: the number of times it has been called and the number of times it has been overridden by a subclass. Each relationship that contributed to a popularity value can be recovered after the fact; this traceability is important for displaying the results to the user (Section 3.4).

3.3 Querying and browsing

Our prototype tool, called PopCon, supports two main interaction mechanisms: querying and browsing. These enable developers to access information about specific APIs in both a top-down and bottom-up manner [20]. The results are presented in a manner that enables the developer to methodically traverse from one API ele-

ment to the next; this type of investigation was advocated by Robillard et al. as a strategy shared by effective developers [17].

Developers initiate queries using a search form in which they can enter the name of the API they are interested in. This mechanism is useful for developers who know what they are looking for (e.g., is `ASTVisitor` frequently extended). After submitting their query, PopCon returns a list of potential APIs for the developer to select from; if there is only one potential match, PopCon jumps directly to the results page. The results page is described in Section 3.4.

In contrast to querying, the browsing approach is directed at developers who are approaching the framework with less detailed knowledge. When browsing, the developer simply starts with the top-level package results view for the whole framework and drills into it, through successively more-specific packages, until they are satisfied.

3.4 Displaying results

Developers interact with PopCon using a web-based interface; this was chosen to minimize the amount of time required to implement the proof-of-concept and to maximize the flexibility in making changes to the tool while its requirements were in flux. Developers are also familiar with web interfaces; their usage encourages lightweight investigation as the developers would not need to install any new tools.

PopCon displays its results at four different layers of specificity. The layer that is shown in the results view is tied to the specificity of the API being considered (e.g., a package, a class, a field or method, or an example usage). The first layer provides a package-level overview. This overview can be used for any subset of packages (e.g., `org.eclipse` and all sub-packages or the leaf package `org.eclipse.swt.browser`). The second layer provides an overview for any class or interface. In the third layer, the developer can look at the usage associated with any fact for a method or field within a class or interface. Finally, the source encoding that usage can be viewed. Each of these can be navigated using only the mouse; the developer does not need to know specific entity names in advance.

For each of the first three layers, the results are displayed using a series of horizontal bar charts (e.g., see Figure 1). These charts provide the name of the API and use a bar to represent its popularity relative to the other APIs on the chart. These graphs provide a simple, light-weight mechanism by which the developer can glance at the results page and quickly get a feeling both for the magnitude and variation of the values they display. By clicking on any of the bars in the charts the developer can change the level of detail they are seeing, either broadening their investigation or narrowing it.

4. TASKS

PopCon can be used both by API producers and API consumers. While both of these groups can use the popularity measure in their tasks, they use it in different ways. API producers are interested in their own APIs and how they are being used while consumers want to discover which APIs perform important functions within the framework and, in turn, to figure out how to use them. In this section we describe four tasks for which PopCon can be used to help the developers described in our motivational scenario (Section 2).

4.1 API Producer

API producers have an imperfect view of how their APIs are being used. They create APIs and release them to their clients; consumers can provide feedback on the APIs using email, newsgroups, or bug reports. While these mechanisms are sufficient for reporting problems with the APIs, they do not provide the API producer with a clear picture of how their APIs are being used. An absence of bug reports can mean that an API is not being used, or is being used but not in the way it was intended, or is being successfully used in the way it was intended. From the API producer's point of view, PopCon serves to provide richer feedback from their APIs' consumers.

4.1.1 How should I prioritize my efforts?

API developers operate in a time-limited environment; there are always many bugs to fix and not enough time to fix them all. PopCon can help these developers triage their bugs by prioritizing those that correspond to APIs that are frequently used. Conversely, a developer may choose not to fix bugs on APIs that are infrequently used. For example, consider the API producer who owns all the APIs in the `org.eclipse.swt.browser` package and who wants to know their frequency of use. Starting on the package overview page, the developer queries PopCon using the package name she is interested in; PopCon computes and returns the package-level popularity report for that package. One of the returned graphs is shown in Figure 1; using this, the developer sees that several methods in the `Browser` class are called the most often, as expected since she designed the API this way. As she has some bugs to fix in `Browser`, she navigates to its PopCon page; this page provides complete graphs showing which of its methods have been overridden, which methods have been called, and which fields have been referenced. Here the developer notices something curious: 8 API methods are never used by any clients. Six of these handle removing listeners; because of this, she updates their corresponding `addListener` documentation to remind developers to de-register listeners when they are done with them.

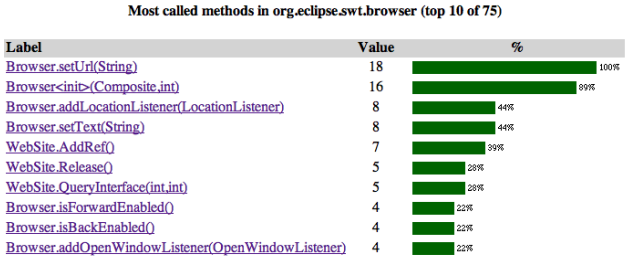

Figure 1: Most called methods in `swt.browser`

4.1.2 How is my API being used?

In addition to knowing that their API is being used by other developers, the producer would like to know that it is being used for tasks that it was intended to support. Furthermore, she would especially like to know if it was being used in ways she did not anticipate or did not intend. In these cases, she may wish to update the documentation to make it clear what the intended use of the API is, or to create a new API to support developers who were using the API incorrectly but for a purpose that the framework could support. The API owner for the `Browser` class can see how any of its methods are being used by its clients, by selecting a method from the `Browser` PopCon page. PopCon then returns the source code for that usage for the developer's perusal.

4.2 API Consumer

API consumers have a broader interest than the API producer did. They are not experts in the framework as they did not create it themselves. They also cannot directly manipulate the framework to do what they want: either the functionality they want to access

is available to them in an API they can use, or they must create the functionality themselves within their own system. For large frameworks, such as Eclipse, the scope of the provided API can be overwhelming (Eclipse Europa consists of 10,000+ public, non-internal, classes and interfaces and 95,000+ methods).

4.2.1 What APIs should I investigate?

To help API consumers manage the complexity of a large number of APIs, PopCon can direct them to those APIs that have been relevant to the most other developers; at the broadest level this means the most popular API in the whole system. From there they can narrow their view to packages that seem most relevant to their task (e.g., in our case, the `browser` package). Looking at the same initial view of the package that the API producer looked at, the developer notices that `Browser` is by far (16 to 1) the most instantiated type in the package (Figure 2). After checking the JavaDoc, the developer decides to continue investigating the `Browser` class. Looking at the overview page in PopCon for `Browser`, the developer notices from one of the graphs (the frequency of method calls, Figure 3) that two methods comprise 71% of the call targets on the `Browser` class. This helps them to avoid investigating all 28 public API methods and focus just on a select few. Going back to the `browser` package overview, the developer also notices from the most implemented interfaces graph that `LocationListener` and `ProgressListener` are the most implemented interfaces in the package, occurring far more often than the other 6 `Listener` interfaces. In this scenario, PopCon has helped the developer to discover those types and methods that are most frequently used in the `browser` package, helping them to avoid manually investigating the entire framework.

Most instantiated types from org.eclipse.swt.browser (top 7 of 7)

Label	Value	%
Browser.<init>(Composite,int)	16	100%
WindowEvent.<init>(Widget)	1	6%
ProgressEvent.<init>(Widget)	1	6%
LocationEvent.<init>(Widget)	1	6%
TitleEvent.<init>(Widget)	1	6%
StatusTextEvent.<init>(Widget)	1	6%
WebSite.<init>(Composite,int,String)	1	6%

Figure 2: Most instantiated types in `swt.browser`

Most called methods on org.eclipse.swt.Browser (top 20 of 20)

Label	Value	%
Browser.setUrl(String)	18	100%
Browser.(Composite, int)	16	89%
Browser.setText(String)	8	44%
Browser.addLocationListener(LocationListener)	8	44%
Browser.isForwardEnabled()	4	22%
Browser.isBackEnabled()	4	22%
Browser.addStatusTextListener(StatusTextListener)	4	22%
Browser.stop()	4	22%
Browser.addProgressListener(ProgressListener)	4	22%
Browser.addOpenWindowListener(OpenWindowListener)	4	22%
Browser.addTitleListener(TitleListener)	3	17%
Browser.addCloseWindowListener(CloseWindowListener)	3	17%
Browser.forward()	3	17%
Browser.addVisibilityWindowListener(VisibilityWindowListener)	3	17%
Browser.back()	3	17%
Browser.getUrl()	3	17%
Browser.execute(String)	2	11%
Browser.refresh()	2	11%
Browser.isFocusControl()	1	6%
Browser.removeLocationListener(LocationListener)	1	6%

Figure 3: Most called methods on `Browser`

4.2.2 How do I use this API?

Once the developer has located an API he is interested in, he can then view examples of how other developers have employed that API. As with the API producer, he can view either an abstraction of the usage (class, method, and field names) as well as the concrete usage in the source code. By looking at the source he can infer pre- and post-conditions that other developers have learned about while using that API, which may not be well documented. On the PopCon page for the `Browser` class, the developer can select the initializer for the class and be given the list of methods that call that initializer. He can then click on any of the 16 callers to get the source code showing how that method initialized and configured the `Browser` object.

5. RELATED WORK

Several approaches exist as potential alternatives to API popularity.

Reverse engineering derives design-level information and/or requirements from source code (e.g., [15]), sometimes focusing on visual representations of that information (e.g. [13, 19]). Such approaches fail to alleviate the issue of information overload that API consumers must face, and do not support querying by API producers of existing usages. Reflexion models [16] help with these problems, but require that the developer provide a high-level model and a mapping specification from the source to the high-level model; errors are simple to produce in this activity and difficult to detect.

Other work has focused on how best to inform API consumers of the design intent of the API producers (e.g. [18]). This information sometimes focuses on patterns of intended usage [11, 7] or constraints on usage [10]. Unfortunately, this places a further strain on limited development resources, as such information must be manually constructed and the API producer must pre-conceive all possible usage scenarios. This helps neither API consumers to appreciate what is most important in the sea of available information, nor API producers to understand how their APIs are actually being used.

Usage pattern mining has been applied for such understanding purposes. CodeWeb [14] infers association rules ("if m1 is called, m2 also tends to be called") via frequent pattern mining and supplies a strictly browse-based interface to locate examples. MAPO [22] represents source in a highly abstract form before it also applies frequent pattern mining; as a result, it is able to detect common (simplified) sequences of calls that tend to be made. Neither approach supports the need for locating unusual usages, and neither allows its results to be tailored in a query-based mode (violating the principle that search strategies should not be imposed [20]). In future, we intend to investigate a technique that permits all examples to be hierarchically categorized.

A variety of approaches utilize program databases [5, 13, 9] but generally seek to support as much as information as possible. In contrast, our repository contains a smaller amount of information tailored to the needs of quick searches for examples. This repository is essentially identical to that we have used in our previous work [8], where it was used to find context-relevant examples rather than understanding the usage popularity of APIs.

6. DISCUSSION

Our popularity measure simply enumerates the number of times any particular API is used in in a specific way. We do not contend that this is the only way to measure the popularity of an API; for instance, there may be some way to combine each of the bar charts to get one unified picture of an API's popularity. While we have found that this straightforward approach can effectively highlight

important API it can also be obscured by mundane, but frequent API usage. For a large API, such as Eclipse, API popularity at the broadest level can be misleading. For instance, the most called method in all of Eclipse is `Control.setLayoutData(Object)` in the SWT package. While this method is useful if a developer is creating the UI for a plug-in, knowing that it is often used is not very helpful. However, at a more targeted level, the results can be meaningful. For instance, for the `org.eclipse.jdt.core` package, the most often called methods are `IJavaElement.getJavaProject()` followed by `ASTNode.accept(ASTVisitor)`. The first provides an often-used accessor while the second points to two key types in the JDT, `ASTNode` and `ASTVisitor`.

Going forward, we are investigating techniques to make PopCon's presentations more useful. The current measure gives a strict numerical indication of the number of times a program element is used. By analyzing the usages themselves, we aim to categorize the individual results into "buckets". These buckets would group common API usages together. For instance, while `Control.setLayoutData(Object)` is called thousands of time in Eclipse, it may only be used in a handful of different ways (e.g., describing the UI in a dialog, a wizard, a form, and a preference page). Developers could then investigate these buckets, before looking at lists of results. We will also evaluate our approach with both Eclipse API producers and API consumers.

7. CONCLUSIONS

In this paper, we have presented PopCon, a prototype tool that can help API producers understand how their APIs are being used as well as help API consumers locate important API and get examples of how it is used in practice. PopCon calculates a popularity measure for every API in a framework by analyzing its static structure. We have presented four sample tasks describing how this measure can be used by both API producers and API consumers. This kind of approach is especially relevant to the Eclipse project as Eclipse constitutes a broad set of APIs that are actively being maintained and upgraded, and are being used by development teams around the world. PopCon can help reduce the effort required of plug-in developers to discover important API; it can also help Eclipse core developers better understand how their APIs are being used in production environments.

8. ACKNOWLEDGMENTS

This work was supported by a Collaborative Research and Development Grant from the Natural Sciences and Engineering Research Council of Canada and IBM Canada.

9. REFERENCES

[1] J. Bosch, P. Molin, M. Mattsson, and P. Bengtsson. Object-oriented framework-based software development: Problems and experiences. *ACM Computing Surveys*, 32(1es):3, 2000.

[2] D. Brugali, G. Menga, and A. Aarsten. The framework life span. *Communications of the ACM*, 40(10):65–68, 1997.

[3] S. Demeyer, T. D. Meijler, O. Nierstrasz, and P. Steyaert. Design guidelines for "tailorable" frameworks. *Communications of the ACM*, 40(10):60–64, 1997.

[4] J. des Rivières. Evolving Java-based APIs. `http://wiki.eclipse.org/index.php/Evolving_Java-based_APIs`, 14 February 2007. Revision 1.1.

[5] P. Devanbu, R. Brachman, and P. G. Selfridge. LaSSIE: A knowledge-based software information system. *Communications of the ACM*, 34(5):34–49, 1991.

[6] M. Fayad and D. C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[7] G. Froehlich, H. J. Hoover, L. Liu, and P. Sorenson. Hooking into object-oriented application frameworks. In *Proceedings of the International Conference on Software Engineering*, pages 491–501, 1997.

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

[9] D. Hou and H. J. Hoover. Source-level linkage: Adding semantic information to C++ factbases. In *Proceedings of the International Conference on Software Maintenance*, pages 447–458, 2006.

[10] D. Hou and H. J. Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[11] R. Johnson. Documenting frameworks using patterns. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 63–76, 1992.

[12] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(5):22–35, 1988.

[13] A. Mendelzon and J. Sametinger. Reverse engineering by visualizing and querying. *Software: Concepts and Tools*, 16:170–182, 1995.

[14] A. Michail. CodeWeb: Data mining library reuse patterns. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 827–828, 2001.

[15] H. A. Müller and K. Klashinsky. Rigi: A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.

[16] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[17] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering*, 30(12):889–903, 2004.

[18] C.-H. Shih and B. Anderson. A design/constraint model to capture design intent. In *Proceedings of the 4th ACM Symposium on Solid Modeling and Applications*, pages 255–264, 1997.

[19] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 11th International Conference on Software Maintenance*, page 275, 1995.

[20] M.-A. D. Storey, K. Wong, and H. A. Müller. How do program understanding tools affect how programmers understand programs? *Science of Computer Programming*, 36(2–3):183–207, 2000.

[21] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.

[22] T. Xie and J. Pei. MAPO: Mining API usages from open source repositories. In *Proc. of the International Workshop on Mining Software Repositories*, pages 54–57, 2006.