UNIVERSITY OF CALGARY

Refactoring References for Library Migration

by

Puneet Kapur

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA
July, 2010

© Puneet Kapur 2010
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Refactoring References for Library Migration” submitted by Puneet Kapur in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE.

Supervisor, Dr. Robert J. Walker  
Department of Computer Science

Dr. Jörg Denzinger  
Department of Computer Science

Dr. Ron George  
Haskayne School of Business

Mr. Ron Murch  
Haskayne School of Business
Abstract

Dealing with change is a persistent problem that has vexed software developers and preoccupied the discipline of software engineering for more than 30 years. Changes to software can occur for a variety of reasons, and can originate from a variety of sources; developers may change their own internal source code, respond to changes in external software upon which their own software depends or introduce changes to software they write that is then reused by other software developers.

The origin of a change impacts the manner and the tools with which developers have to deal with it. When making alterations to their own source code, developers can use automated tools known as refactorings to enact changes. Existing refactorings rely on developers having access to the source code where an entity is declared and the locations where it is referenced. Refactoring tools ensure that changes to a declaration are propagated to update all the locations where that entity was referenced.

However, there exist situations in which a declaration is not available for refactoring. When creating applications that make use of external software, developers make references within their source code to entities declared by—and whose source code is managed by—someone else. When this third-party software changes, the references within the developer’s own source code are broken, an outcome we refer to as dangling references.

We investigate the problem of dangling references through a detailed study of three open source libraries. We find that the introduction of dangling references during library migration is a significant real problem, and characterize the specific issues that arise. Based on these findings we provide and test a prototype tool, called Trident, that allows programmers to refactor references. Our results suggest that supporting the direct refactoring of references is a significant improvement over the state-of-the-art.
Acknowledgments

Redacted from Online version
Redacted from Online Version
Redacted from Online Version
Dedication

To my Grandfather, Vishwanath Kapur विश्वनाथ कपूर we never met but your sacrifice shaped my life.

To my Father, Prabhat Kapur प्रभात कपूर who embodies the ideals my Grandfather stood for and all that I hope to become.

To my Mother, Anita Kapur अनिता कपूर for having the wisdom and patience to teach me all of the above.
Table of Contents

Abstract ................................................................. ii
Acknowledgments ...................................................... iii
Dedication ............................................................... vi
Table of Contents ....................................................... vii
List of Tables .......................................................... ix
List of Figures .......................................................... x
1 Introduction .......................................................... 1
  1.1 API Evolution .................................................... 2
    1.1.1 Managing API Evolution .................................. 5
  1.2 Tool Support ..................................................... 6
    1.2.1 Refactoring Tools ......................................... 7
  1.3 Research Approach ............................................. 10
  1.4 Thesis Statement ............................................... 11
  1.5 Overview of Related Work .................................... 11
  1.6 Thesis Structure ............................................... 13
2 Motivation .......................................................... 15
  2.1 Binary Incompatibilities Arise ................................ 15
  2.2 How Could the Task Have Been Achieved? .................... 16
  2.3 Following Up with the Developers Involved ................. 19
  2.4 Summary ......................................................... 20
3 API Change in the Wild ............................................ 22
  3.1 System Selection ............................................... 22
  3.2 Study Design ................................................... 24
  3.3 Predicting API Change ......................................... 25
  3.4 Study Results .................................................. 28
    3.4.1 High-Level Results ....................................... 29
    3.4.2 Low-Level Changes ....................................... 30
  3.5 Summary ........................................................ 33
4 Refactoring References: Prototype Tooling ....................... 41
  4.1 A Consideration of the Required Features .................... 41
  4.2 Goals ........................................................... 47
  4.3 Application and Features ...................................... 48
    4.3.1 Selection .................................................. 51
    4.3.2 Search Configuration ..................................... 51
    4.3.3 Restriction ............................................... 52
    4.3.4 Refactoring Configuration ................................ 52
    4.3.5 Preview and Enactment .................................. 53
  4.4 Tool Implementation ........................................... 53
    4.4.1 Selection .................................................. 53
    4.4.2 Search Configuration ..................................... 55
    4.4.3 Restriction ............................................... 56
4.4.4 Refactoring Configuration ............................................. 57
4.4.5 Preview and Enactment ................................................. 59
4.5 Summary ........................................................................ 59
5 Case Studies ..................................................................... 60
5.1 Case Study Methodology .................................................. 60
5.1.1 Systems .................................................................... 61
5.1.2 Task ......................................................................... 62
5.1.3 Setup ......................................................................... 62
5.2 Participant 1 ................................................................... 64
5.2.1 Tool Treatment .......................................................... 64
5.2.2 Manual Treatment ....................................................... 67
5.3 Participant 2 ................................................................... 71
5.3.1 Tool Treatment .......................................................... 71
5.3.2 Manual Treatment ....................................................... 73
5.4 Results and Analysis ....................................................... 76
5.4.1 Research Question #1 ................................................. 76
5.4.2 Research Question #2 ................................................. 78
5.5 Summary ........................................................................ 78
6 Discussion ........................................................................ 81
6.1 Threats to Validity ........................................................... 81
6.2 Tool Limitations ............................................................... 84
6.3 Usability ......................................................................... 84
6.4 Future Work .................................................................... 85
6.5 Summary ........................................................................ 88
7 Related Work .................................................................... 89
7.1 Related Work on API Change ........................................... 89
7.2 Related work on API Migration Tool support ......................... 93
7.2.1 Semantically-Based tools ............................................. 94
7.2.2 Migration Tools: Formal Change Specifications ............... 94
7.2.3 Migration Tools: Change Recording and Replay ............. 97
7.2.4 Migration Tools: Custom Adaptation Code .................... 97
7.2.5 Migration Tools: API Recommenders .......................... 98
7.3 Summary ........................................................................ 99
8 Conclusion ....................................................................... 100
8.1 Contributions .................................................................. 101
8.2 Future Work ................................................................... 102
Glossary .............................................................................. 103
Bibliography ........................................................................ 105
A Ethical Review and Informed Consent ................................. 115
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Change cases involved in the Chartwell library migration scenario.</td>
<td>17</td>
</tr>
<tr>
<td>3.1</td>
<td>Top 40 search queries from FindJar.com.</td>
<td>23</td>
</tr>
<tr>
<td>3.2</td>
<td>API breakages in the studied systems.</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>Binary incompatibilities between versions of HTMLUnit.</td>
<td>35</td>
</tr>
<tr>
<td>3.4</td>
<td>Binary incompatibilities between versions of JDOM.</td>
<td>36</td>
</tr>
<tr>
<td>3.5</td>
<td>Binary incompatibilities between versions of log4j.</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>Raw counts of deprecated entities and total binary incompatibilities between versions of HTMLUnit.</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>Raw counts of deprecated entities and total binary incompatibilities between versions of JDOM.</td>
<td>39</td>
</tr>
<tr>
<td>3.8</td>
<td>Raw counts of deprecated entities and total binary incompatibilities between versions of log4j.</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>The full set of changes needed to port JaxMe from log4j version 1.2.8 to version 1.3.</td>
<td>63</td>
</tr>
<tr>
<td>7.1</td>
<td>Systems studied for API change by Dig and Johnson [2006].</td>
<td>89</td>
</tr>
</tbody>
</table>
List of Figures

1.1  An example for understanding different kinds of compatibility. . . . . . . . . . 5
1.2  An example for which the Rename Method refactoring is to be applied. . . . . 8
1.3  An example for which the Rename refactoring has been applied. . . . . . . . . 9

4.1  Screenshot of `getParent()` dangling reference. . . . . . . . . . . . . . . . . . . 42
4.2  The full specification required by the method of Balaban et al. [2005] to mi-
    grate from `Vector` to `ArrayList`. . . . . . . . . . . . . . . . . . . . . . . . . 46
4.3  Trident search configuration. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.4  Trident search results. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.5  Trident refactoring configuration. . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.6  Trident reference refactoring preview. . . . . . . . . . . . . . . . . . . . . . . 50
4.7  Selection assistance activated by attempting to select the argument list of the
    method. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.8  Selection assistance completed. . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.9  `SimpleType` and `QualifiedType` refactoring configuration dialog. . . . 58
Chapter 1

Introduction

“It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.”

—Megginson [1963, p. 4]

“Man is a tool-using animal ... Without tools he is nothing, with tools he is all.”

—Carlyle [1831, Bk. 1, Ch. 5]

Software experiences constant environmental pressure to change [Belady and Lehman, 1976, Parnas, 1994]. The environmental pressures experienced by software have been recognized as falling into four large categories [Lientz and Swanson, 1980]: the need to repair faults, the need to extend functionality, the need to operate in novel environments, and the need to improve internally.

As software has grown in complexity, so has the complexity of the environment in which it is developed. A lone programmer working in isolation need only worry about changes he makes to his own source code; apart from computer science students working on last-minute assignments, rarely does such a simplified working environment exist. Software developers on any realistic project must also worry about changes being made simultaneously by colleagues [Kraut and Streeter, 1995, Grinter, 1996, Sarma et al., 2003, Gross and Prinz, 2003, O’Reilly et al., 2005, Biehl et al., 2007, Bertram et al., 2010] and changes made to software pieces provided by external organizations on which their own software depends [Sarma and van der Hoek, 2006, Holmes and Walker, 2010]. The need to cope with such changes explains why developers have sought—and continue to seek—better software tools to carry out their jobs.
1.1 API Evolution

The general problem of a change in one part of a software system causing problems in another part of the system has plagued software engineers since the 1970s. Britton et al. [1981] state:

It is a common but undesirable property of embedded software that a change in a device interface requires widespread changes to the software because many programs are based on arbitrary interface details. If an interface changes, programs depending on it become invalid.

In the modern context, third-party software packages are often termed libraries (dynamically-linked libraries (DLLs) in C/C++ or Java archive (JAR) files in Java)\(^1\) in reference to the fact that within each package a library of functionality is provided for software developers to reuse. The books, periodicals, microfiche, etc. of a physical library having been replaced with classes, methods, and fields from object-oriented programming.

Once a library catalogue is published, librarians rely on it as the definitive guide of what resources are available. Similarly with software libraries the application programming interface (API) is the authoritative guide of what features and functionality are available. If a book is misfiled, reclassified, or lost then all the locations in the library catalogue making reference to it need to be updated. In software libraries if a class/method/field is renamed, moved, or deleted then the code which relied upon that library must be updated so every reference to modified entities in the old API is substituted with one or more references to the appropriate replacement(s) from the new API.

Library designers are encouraged to create interfaces (APIs) with due deliberation to eliminate the need for future changes [Parnas, 1972, Parnas et al., 1984]. Yet even in the unlikely event that an API is perfect at inception, software environmental pressures ensure that it does

---

\(^1\)Except where otherwise stated, all source code references within this thesis are written in the Java programming language.
not stay that way for long. As Ungar stated in an article by Pancake [1995], “the notion of interface is much more of an illusion than people give it credit for . . . every honest programmer will admit there are times when you have to change your mind, and the interface changes.”

API changes may be inevitable but not all API changes need to be addressed in the same way. Addition of classes, methods, or fields to an API is usually free of complications since client code, as yet, contains no references to these entities. In restricted cases the movement of methods and fields may also be benign. In general “as long as there is no change in a library’s syntax or semantics, applications can use update[d] libraries simply by importing and linking the new version” [Chow and Notkin, 1994, p. 359].

Library changes that continue to operate with client code written against the previous library version are referred to as backwards-compatible changes or non-breaking changes [Dig and Johnson, 2006]; conversely library changes that are not backwards compatible are called non-backwards compatible changes or breaking changes. When a breaking change occurs, client code built with an older version of the library “may fail to compile or link [with the newer library version]. Or, the application might compile fine but behave differently at runtime” [Dig and Johnson, 2006, p. 80].

Within this concept of a breaking change are subtle yet distinct notions of how breaks arise. An API may be backwards incompatible for deficiencies in one of the following three categories [Tulach, 2008]:

1. **Binary Compatibility**: Two versions of the same library are binary compatible if any program written and successfully compiled into appropriate binary form against an older version of the library, can link its binary form with the newer version.

2. **Source Compatibility**: Two versions of the same library are source compatible if any program written and successfully compiled against the older version can also be success-

---

2In some relatively rare cases, addition of such entities may cause client code to fail compilation. For instance the addition of a method to an interface requires recompilation.
fully compiled against the new version.

3. **Functional Compatibility**: Two versions of the same library are *functionally compatible* if any code written, compiled, linked, and executed against the older version can successfully be executed against the new version to produce the same result.

In particular the Java Language Specification [Gosling et al., 2005] takes pains to note that “a set of compatible binaries can be produced from sources that will not compile together”. However, the inverse of this statement also holds true; a set of incompatible binaries cannot be produced from source that will compile together. In other words, if binary incompatibilities exist between library versions then source incompatibility also exists. A client program written and successfully compiled against the old library version will require source code changes prior to being successfully compiled against the new library version. More formally, binary compatibility is a necessary but not sufficient condition for source compatibility and for functional compatibility; source compatibility is neither necessary nor sufficient for functional compatibility.

We use the example in Figure 1.1 to illustrate the differences between types of compatibility. Consider moving from the Wine interface version 1.0 to version 1.1 by adding a new method `public String getRegion();` all the pre-existing binaries\(^3\) compiled against Wine 1.0 were and can continue to link and run with the new code. However, *source code* that migrates to the new version and is compiled against Wine 1.1 will break. In this case, the class `RedWine` would have to implement a `getRegion()` method to successfully compile. Thus the addition of a method to an interface is source incompatible but binary compatible. Conversely if we remove the `getPrice()` method then previously compiled code that relied on the existence of this method will fail. Deletion of an interface method is binary incompatible.

\(^3\)In Java, binaries are often referred to as the *bytecode*. 
public interface Wine {
    public int getPrice();
    public int getYear();
}

public class RedWine implements Wine {
    public int getPrice() { return 155; }
    public int getYear() { return 1978; }
}

Figure 1.1: An example for understanding different kinds of compatibility.

To the best of our knowledge, no one has previously used binary incompatibility as a criterion to measure library changes. Other work [Dig and Johnson, 2006, Xing and Stroulia, 2006] has focused exclusively on changes in terms of source and/or behavioural incompatibility. Admittedly such work is important since source incompatibility prevents developers from being able to compile with the newer library. However, not all production environments permit complete recompilation of source code whenever a library update occurs (e.g., dynamic software updating [Orso et al., 2002, Gharaibeh et al., 2007]). The Java language itself attempts to guarantee binary compatibility but does not strive for source compatibility.\footnote{\url{http://java.sun.com/javase/6/webnotes/compatibility.html#source} [last accessed 28 April 2010].}

As such, studies of library change at the level of source incompatibility can be considered an \textit{upper bound} measure for the library change that is more relevant in environments where full recompilation is possible. In contrast, viewing change at the level of library binary incompatibility can be considered a \textit{lower bound} measure of library change. Any changes at this level will affect all library users and necessitate source code modifications.

1.1.1 Managing API Evolution

There are provisions within the Java language and software development conventions to notify developers of impending API changes thus allowing them to plan for an orderly transi-
tion. Java allows library designers to mark part of the API as obsolete—with the potential for removal without further warning—by using the “deprecation” feature which is implemented using Javadoc tags.

Javadoc is a tool supported by the Java programming language that permits detailed documentation to be placed directly in the source code in the form of specialized comments, called document comments; the tool can then generate HTML documentation from these comments and the structure of the source code. Tags within the document comments allow cross-references and metadata to be identified; these are prepended with the “@” symbol. For example, the @since tag is used to indicate details of when a particular portion of source code was added to the API. The @deprecated tag indicates that an entity “is no longer important … since it has been superseded and may cease to exist in the future”. However deprecation is limited to signalling future changes to source code entities that already exist and it relies on developers reading the Javadoc shipped with the new library.

A different mechanism is needed to signal binary level changes and quickly advertise other changes without relying on developers to read all the documentation. A popular solution is to adopt a library numbering convention where three digits are used to indicate major, minor and very specific changes. Most technology users are already familiar with this idea; it is understood that moving from version 1.0.0 of some software to 1.1.0 should involve minor change but moving to version 2.1.0 could involve breaking changes.

1.2 Tool Support

Earlier we noted that API evolution involves developers having to deal with changes arising in source code that is outside their control. A simple API change like renaming a method from putValue() to setValue() will cause all client programs to break. Yet developers

\footnote{http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/deprecation/deprecation.html#javadoc_tag [last accessed 28 April 2010].}
routinely carry out such changes to their own code with little difficulty [Ko et al., 2005, Murphy et al., 2006, Murphy-Hill et al., 2009].

The difference is that when developers make such changes to their own source code they have complete access to (1) the declaration of the method (or other source code entity) and all the references to that method, which means that (2) they can take advantage of automated refactoring tools to implement the change. These two points are central themes within the following discussion.

1.2.1 Refactoring Tools

Refactoring refers to the process of changing the structure of a program—to improve quality attributes such as understandability, maintainability, extensibility, etc.—without changing its behaviour [Xing and Stroulia, 2006]. The relevance of Refactoring to this work is perhaps not surprising given that the term was coined by Opdyke [1992] based on his study of software evolution with the Choices operating system. Fowler’s book [Fowler, 1999] on the subject of refactoring popularized the term and provided a catalogue of common refactorings that programmers could implement.

Automated refactoring is now a key feature of modern integrated development environments (IDEs) [Opdyke and Johnson, 1990, Opdyke, 1992, Griswold and Notkin, 1993]. The popular Eclipse IDE\textsuperscript{6} provides about 25 default refactorings.\textsuperscript{7} For example [as argued by Fowler, 1999], let us assume the class Account and the class ReconcileAccount are within the developer’s code base (see Figure 1.2). The developer notes that the setUpdat-ed() method on line 10 is poorly named and decides to rename it to setTimeStamp() to better convey its actual function. With the cursor placed on the setUpdated() method name he invokes the Rename refactoring and provides setTimeStamp() as the new name.

The revised code is shown in Figure 1.3. Notice that by changing the method name at

\textsuperscript{6}http://www.eclipse.org [last accessed 28 April 2010].

\textsuperscript{7}As Eclipse is highly extensible, this number could be much higher in practice.
public class Account {
    private int _timeStamp;
    public Account() {};

    public Account(int timeStamp) {
        setUpdated(timeStamp);
    }

    public void setUpdated(int arg) {
        _timeStamp = arg;
    }
}

public class ReconcileAccounts {
    private List allAccounts;

    public void runBatch(int timeStamp) {
        for(Account account : allAccounts) {
            account.setUpdated(timeStamp);
        }
    }
}

Figure 1.2: An example for which the Rename Method refactoring is to be applied.

line 10, the refactoring has also located and updated the invocations of that method at line 7 and line 20 to reflect the new name. This simple refactoring shows that changing the declaration of the method name causes all the references to be updated.⁸

The correspondence between a declaration and its reference is made possible via resolved information about its corresponding declaration, called bindings within Eclipse. By default the Eclipse compiler uses syntactic details of the language to generate a tree representation of the source, called the abstract syntax tree (AST). In Figure 1.2, the AST provides us with enough information to know that _timeStamp at line 11 is an identifier that is part of an assignment

---

⁸In fact Eclipse allows you to change the method name by selecting a declaration or a reference that it is able to unambiguously resolve to a declaration. However this does not alter the salient point that refactoring relies on access to declarations, at a minimum, to proceed. Also note that Balaban et al. [2005] used the terms declarations and call/allocation sites to describe the same idea.
public class Account {
    private int _timeStamp;

    public Account() {};

    public Account(int timeStamp) {
        setTimeStamp(timeStamp);
    }

    public void setTimeStamp(int arg) {
        _timeStamp = arg;
    }
}

public class ReconcileAccounts {
    private List allAccounts;

    public void runBatch(int timeStamp) {
        for (Account account : allAccounts) {
            account.setTimeStamp(timeStamp);
        }
    }
}

Figure 1.3: An example for which the Rename refactoring has been applied.

statement. Bindings supplement this information by telling us that _timeStamp is a variable of type int and that it is associated with the declaration at line 2.

Let us contrast this situation with what occurs in the case of API migration by reusing the same example but this time the Account class (lines 1–13 in Figure 1.2) is defined in a third-party library and the client code contains the class ReconcileAccounts (lines 15–22). As before the library designer performs a Rename refactoring on the library code to change the declaration of setUpdated() to setTimeStamp() which also transforms the reference at line 7. When this new JAR file is sent to the library user, the reference to setUpdated() (line 20) is now broken, an outcome that we refer to as a dangling reference.

The reference to setUpdated() is syntactically correct so an AST can still be con-
structed. However the reference no longer corresponds to a declaration in the source code (i.e., it “dangles” freely) and as such no binding information exists. Dig and Johnson [2006, p. 94] summarize this situation as follows:

Component designers rename an instance method in the component. They find and update all the callers and implementors of the method to reflect the new name. For the component itself this change is safe and does not modify its behaviour. However, remote applications that call the renamed method are broken. Thus a behaviour-preserving change (refactoring) for the component might lead to a breaking change for the application.

Current refactoring tools are unable to assist in this scenario because (1) they operate on the assumption that a change in the declaration is the principal cause of updates to the references—the notion of refactoring references independently is absent; and (2) on a practical level refactoring tools rely on bindings to disambiguate references to `setUpdated()` from other methods that may share the same name but are unrelated to the dangling references we seek in the source code.

1.3 Research Approach

There exist practical situations in which a declaration is not available for refactoring or would be inappropriate to transform, but instead, only the references should be changed. The direct refactoring of references is not available at present, reducing support to the level of text editing; thus, greater effort is required of developers with an ensuing greater likelihood that they introduce errors.

We see four situations in which the refactoring of references is of potential benefit. (1) In library migration in practice, the developer is likely to remove an old version and insert a new one; if the library’s API has changed, dangling references will now be present in the
developer’s code. (2) In test-driven development [Beck, 2003, Maximilien and Williams, 2003, Erdogmus and Morisio, 2005], test cases are written before their corresponding declarations. As development proceeds, revised design ideas can necessitate the refactoring of the tests even if the declarations have not yet been written. (3) In pragmatic reuse scenarios [Holmes and Walker, 2007, 2008, Holmes, 2008], developers opportunistically locate useful functionality from another system that they copy and modify to operate in their own system. This process of integration often involves the refactoring of references to utilize declarations in the developer’s target system. (4) Practical software development involves the ability to revise design ideas in the midst of implementation. This can involve the presence of dangling references, or different ideas about how to use external APIs requiring references to be refactored.

In all these situations, the transformations must operate (a) in the presence of broken semantics and possibly broken syntax, (b) where the details of transformation can vary between locations, and (c) where the developer is the ultimate arbiter of what makes sense.

We focus here specifically on the library migration scenario.

1.4 Thesis Statement

We hypothesize that tool support that supports the systematic transformation of references would improve the performance of library migration tasks, in terms of time to completion and reduction in errors, relative to state-of-the-practice software tools.

1.5 Overview of Related Work

The problem of migrating clients/users of one API to another has been alluded to for more than 30 years [Parnas, 1978, Lampson, 1983] and in that time there have been numerous attempts at designing software tools to make the task easier [Britton et al., 1981]. Various solutions to the problem fall roughly into three categories: lexical, syntactic and semantic tools.
Most lexical tools are variations on the ubiquitous “grep” family of tools [Wu and Manber, 1992]. The Lexical Source Model Extraction (LSME) tool [Murphy and Notkin, 1996] extends Grep by automatically tokenizing source code, allowing developers to write patterns to match complete tokens rather than worrying about raw character strings involving arbitrary whitespace and escape characters. LSME also allows developers to compose lexical patterns into hierarchial relationships. The TAWK tool [Griswold et al., 1996] builds atop lexical search in a slightly different way. In place of tokens, the tool uses a specification provided by the developer to convert source code into an AST. Lexical patterns are then written to match the text within tree nodes with the additional stipulation that only certain types of syntax nodes match. In either case, developers are required to write regular expressions (which can be notoriously complex to deal with in practice [Walker and Viggers, 2004, Holmes et al., 2006]) and/or prompted for a detailed specification of the language (TAWK) or the relationship between patterns (LSME). Finding a lexical match is only half of the problem: developers must also specify how the match is to be transformed, which can be just as difficult.

Syntax-based approaches can use language structure to distinguish between references and declarations. However, (1) such approaches (e.g., the TXL source transformation language [Cordy, 2006]) require learning a query language approximately equal in complexity to the language being searched; (2) syntax can be (temporarily) broken in practical development situations; and (3) syntax alone does not suffice to disambiguate references to identically-named declarations, which occur in practice.

More sophisticated semantics-based tools also exist as program transformation tools and refactoring languages. As with syntactic tools, both approaches rest on the existence or creation of a detailed formal specification. For instance traditional program transformation tools [e.g., Feather, 1989] generate code by transforming a description of the program written in a formal language. Refactoring languages (see examples in Dig and Cebulla [2008]) are also burdened by the high cost of learning a new language to specify the transformation.
Specifically in the area of API evolution some researchers have examined the changes that occur between library versions in the hope that a better understanding of API evolution will inform better tool design [Xing and Stroulia, 2005b, Dig and Johnson, 2006, Dig et al., 2008]. Various attempts at tool support place the onus on the library designer to chronicle the changes that have occurred in order to convey that information to library users [Chow and Notkin, 1994, Henkel and Diwan, 2005, Perkins, 2005]. Other tools analyze the library code base to infer what changes have occurred and provide replacement recommendations [Xing and Stroulia, 2007, Dagenais and Robillard, 2008] but do not alter the client code. Other tools do update client code provided that the library user can provide a formal specification detailing changes in the old API and how those changes should be remapped into the new API [Yellin and Strom, 1997, Balaban et al., 2005, Nita and Notkin, 2010].

In the end, none of the existing approaches sufficiently solves the problem of library migration.

1.6 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 describes an actual industrial scenario of library migration, illustrating the great difficulty inherent in such tasks in modern practice.

Chapter 3 presents an empirical study into API breakages in a set of industrial software libraries, as an indication of their frequency, severity, and properties. We recorded and analyzed the binary incompatibilities between a large number of versions of three industrially relevant software libraries.

Chapter 4 describes our approach to supporting some of these transitions, as embodied in the Trident tool. Trident provides flexible search-and-replace functionality that uses an exemplar supplied by the developer; it leverages a blend of lexical, syntactic, and semantic clues.
according to the developer’s needs. Trident makes use of partial program analysis [Dagenais and Hendren, 2008] to estimate whether two references refer to the same entity. It allows the developer to preview the set of locations that a query would transform, and how these would be transformed. The developer can either proceed or revise the search criteria or transformation specification.

Trident is evaluated through two in-depth case studies with industrial developers in Chapter 5, each of whom was tasked with a realistic library migration problem. Both developers attempted the task first with our tool support and then with only standard IDE support. Qualitative observations about the relative strengths and weaknesses of the treatments are reported along with the developers’ thoughts and opinions.

Chapter 6 discusses remaining issues and future work.

Chapter 7 considers in detail how our approach differs from existing work and is more appropriate for this practical problem.
Chapter 2

Motivation

Consider an actual scenario of class library migration that unfolded at Chartwell Technology, our industrial research partner. At the time, Chartwell employed approximately 40 software engineers working on over 1 MLOC of Java code. The code made extensive use of the XML parsing facilities provided by the JDOM library, version b9.\(^1\) As Chartwell’s product line matured, the XML processing demands increased, prompting a search for a newer XML class library. The then-new release of JDOM (version b10) seemed like the most appropriate replacement candidate. Following the most direct upgrade path, one developer was assigned to replace JDOM-b9 with JDOM-b10 on his classpath, to ensure that the revised code compiled and passed all unit tests and then commit the changes. In practice this class library migration was not so simple.

2.1 Binary Incompatibilities Arise

A comparison of the library versions reveals that there are 120 binary incompatibilities between the two JDOM versions. Some of these changes are unlikely to have any implications for JDOM users, such as the visibility change of the field `org.jdom.input.SAXHandler.currentElement` from `protected` to `private`. While other, seemingly trivial, changes had a considerable impact on the compilation of the Chartwell codebase. For instance, the method `Element.getParent()` was removed and replaced with `Element.getParentElement()`. This method was invoked throughout the code base and its absence alone resulted in 140 compilation errors. Similarly the pretty printing of XML documents was pre-

\(^1\) Although a beta-version, JDOM was the cutting edge technology of the day; alternative libraries were not considered viable by Chartwell.
viously accomplished by instantiating `XMLOutputter` thus: `new XMLOutputter("", true)`. In the new version this task was accomplished with a new `Format` class and all the previous method invocations needed to be replaced with `new XMLOutputter(Format.getPrettyFormat())`. This change resulted in 86 compilation errors. Table 2.1 lists the full set of changes and resulting errors in the Chartwell codebase.

In total there arose 467 compilation errors during this library migration, resulting from just 7 of the 120 binary incompatibilities between the JDOM versions. Thus, the potential impact of the library upgrade would have been far worse if Chartwell had made more extensive use of the JDOM API. To complete the JDOM library migration ultimately took almost 2 full days of effort and involved manual modifications of 274 Java files.

2.2 How Could the Task Have Been Achieved?

The difficulty of this solution stands in stark contrast to the apparent simplicity of the underlying API change. Given this discrepancy, it is worth asking: Why was the task attempted manually when there are a host of code modification tools inside of Eclipse? Consider the alternatives that the developer could have pursued.

The first tool that comes to mind for this task is grep. A search for the regular expression “\.getParent()” and replacement with “\.getParentElement()” seems like the most obvious choice for the first API change. Unfortunately this initial impulse is wrong as there are innumerable invocations of methods named `getParent()` in the Chartwell codebase that correspond to method declarations on different types unrelated to JDOM. The prevalence of duplicate method names can be seen in a lexical search of the Eclipse 3.5.1 libraries—1,233 classes with 3,024 `getParent()` references to a range of different method declarations.

For the moment, let us assume that duplication of method names is a rare event and the
Table 2.1: Change cases involved in the Chartwell library migration scenario.

<table>
<thead>
<tr>
<th>Sample Change</th>
<th>Replacement</th>
<th>Error</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Element.getParent()</code></td>
<td><code>Element.getParentElement()</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>new XMLOutputter(String)</code></td>
<td><code>new XMLOutputter(String, boolean)</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>new XMLOutputter(Format f)</code></td>
<td><code>new XMLOutputter(Format.getPrettyFormat()).outputString(formElement)</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>response = new ResponseXMLFormat(r.getRootElement().detach());</code></td>
<td><code>response = new ResponseXMLFormat((Element)r.getRootElement().detach());</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>if(messages.getChildren().size() &gt; 0) {</code></td>
<td><code>if(messages.getChildren().size() &gt; 0) {</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>XMLOutputter out = new XMLOutputter(Format.getCompactFormat());</code></td>
<td><code>XMLOutputter out = new XMLOutputter(Format.getCompactFormat());</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
<tr>
<td><code>XMLOutputter out = new XMLOutputter(Format.getCompactFormat());</code></td>
<td><code>XMLOutputter out = new XMLOutputter(Format.getCompactFormat());</code></td>
<td><code>-</code></td>
<td><code>+</code></td>
</tr>
</tbody>
</table>
only references in the code base to `getParent()` correspond to the pertinent ones in JDOM. Even so, grep would still not be up to the task. A closer examination of the API change reveals that originally 6 JDOM classes declared `getParent()` methods. Of those, 5 now share a common parent, the new `Content` class, from which they inherit `getParentElement()`.

The 6th class is `Attribute` and it remains outside the new supertype hierarchy. As such `getParent()` invocations on variables of type `Attribute` should not be refactored. Making the necessary syntactic distinctions between such cases is outside the ability of lexical tools such as grep.

Conveniently, Eclipse provides syntactic search support through Java Search (we distinguish Java Search from Eclipse File Search, which is purely lexical) which might prove useful. Initial attempts at specifying `Attribute.getParent()` in our Java Search query appear to work. None of the `getParent()` references unrelated to JDOM appear in the search results. However all the `getParent()` references to JDOM are returned whether they are invoked on `Attribute` variables or on any of the 5 other classes. Refusing to become discouraged we try applying the same strategy to the next API change. Occurrences of `new XMLOutputter("")` need to be replaced with `new XMLOutputter(Format.getRawFormat())` while those of `new XMLOutputter("", true)` need to be replaced with `XMLOutputter(Format.getPrettyFormat())`. Unfortunately Java Search is still not up to the task as it is unable to distinguish between overloaded versions of the same dangling method reference.

An optimist might argue that despite its shortcomings, Java Search has done enough by returning a list of method references and associated filenames that we can perform a replace operation on. We are stymied yet again as Java Search offers no replace option. Regardless, what we need is not `replace functionality` but `refactoring functionality`. Eclipse refactorings provide numerous error checking and convenience features that potentially simplify the user experience. For instance when refactoring `new XMLOutputter("")` to `new
XMLOutputer(Format.getRawFormat()) the appropriate import needs to be added to the affected classes. Similarly when changing method names, refactoring tools ensure the proposed name does not conflict with another name in the same scope. While such points may seem trivial and straightforward, we will see the great pain that they can cause in practice.

2.3 Following Up with the Developers Involved

Over the course of this research we contacted five Chartwell developers (identified as 1 to 5) for other examples of problematic class library migration and informal feedback about how they tackled the problem currently and what they might envision in a solution. From this series of email exchanges and in-person discussions some interesting points emerged.

Developer 1, who at the time was a member of the Build Team, stated “we hold off [on updating library JAR files] as long as possible because updating the code is a pain”. The same developer also pointed out that Chartwell made use of 42 external libraries which had collectively been updated 77 times. Moreover such high reliance on external code, in his experience, was fairly typical for industrial systems.

Developers 2, 3, and 5 all had first-hand experience with dangling references following library updates. Both developers 2 and 3 made use of a commercial library for payment processing that was updated by the vendor to provide new security functionality. The complete details of the API change are unavailable but Developer 2 described the change as

...the method getBalance(int, int) was called by a number of classes.

The addition of one more parameter to this method required finding all the calling classes and modifying them to pass a third integer. Once getBalance(int, int) was updated to getBalance(int, int, int) [by upgrading the JAR file] I used Eclipse to compile all the Java classes in the source. Eclipse would then show me [all the classes] that failed during the compiling. That’s how I found out
all the classes that need to be changed.

If support for refactoring of references was available it would be straightforward to update all the `getBalance(int, int)` calls to include a default third parameter and spare the developer from repetitively altering dozens of java files.

Developer #5 encountered a slightly different use case for refactoring references. He was altering a sub-project within the Chartwell code base to use a new database library. Invocations of factory methods on the old database library (e.g., `getConnection(String url, int timeout)`) needed to be replaced with similarly named methods—with similar parameters—on the new database library (e.g., `initDBConnnection(int timeout, String url)`). This example is particularly interesting because the original library and the replacement library were both present (i.e., complete binding information) yet the changes needed could not be accomplished with standard refactorings. When asked what tools he had used to enact the change, developer #5 stated, “I did it all manually because there is no feature that lets you re-point a method reference to something else”.

2.4 Summary

The experiences at our industrial partner, Chartwell Technology, show that migrating an industrial sized code base to a new library version is an arduous undertaking that is poorly supported by current tools. Moving from version b9 to b10 of the JDOM XML parsing library caused 467 compilation errors in the Chartwell code base that arose from just 7 of the 120 binary incompatibilities between the two library versions. The potential impact of the library upgrade would have been far worse if Chartwell had made more extensive use of the JDOM API. To complete the JDOM library migration ultimately took almost 2 full days of effort and involved manual modifications of 274 Java files.

Comments and discussions with other Chartwell developers emphasized that (a) industrial
systems make extensive use of external libraries and each of these libraries has multiple versions over its lifetime; (b) even minor changes (e.g., renaming) can cause significant problems for library users; and (c) support for refactoring of references could also be useful in situations other than API migration.
Chapter 3

API Change in the Wild

While various authors have mentioned the existence of API changes in practice, no in-depth study exists to point to the extent, quality, and severity of the problem. To this end, we present a systematic investigation of API changes in practice.

We conducted a detailed study of binary incompatibility in three open-source software systems, the results of which bolster some findings by other researchers. In particular, we compare our results to the smaller study of API refactoring conducted by Dig and Johnson [2006] and the follow-on study of refactoring within the Eclipse code base carried out by Xing and Stroulia [2006]. In addition, we examine to what extent developers use, or do not use, the various deprecation features of Java to provide notice of API change.

3.1 System Selection

We sampled the growing body of open-source systems available on the Internet to select three systems to represent API change in general; we chose HTMLUnit, JDOM, and log4j. We argue that these systems are representative of API change according to several criteria. Each system is independent with respect to its application domain; HTMLUnit is used for unit testing, JDOM for XML parsing, and log4j provides application logging functionality. Moreover, we used to two techniques to establish that these systems are industrially relevant Where possible, we obtained download statistics for each system and then supplemented this with internal data (see Table 3.1) from a popular website, FindJar.com, that offers a searchable index of third-party JAR files. When confronted with a missing class file, developers can submit the class name to FindJar.com and it returns a list of candidate JAR files containing that class.
<table>
<thead>
<tr>
<th>Rank</th>
<th>Search Term</th>
<th># Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>org.slf4j.impl.StaticLoggerBinder</td>
<td>24,720</td>
</tr>
<tr>
<td>2</td>
<td>Exception</td>
<td>18,747</td>
</tr>
<tr>
<td>3</td>
<td>org.apache.commons.logging.LogFactory</td>
<td>17,422</td>
</tr>
<tr>
<td>4</td>
<td>javax.xml.stream.XMLStreamException</td>
<td>15,309</td>
</tr>
<tr>
<td>5</td>
<td>org.slf4j.LoggerFactory</td>
<td>13,322</td>
</tr>
<tr>
<td>6</td>
<td>javassist.util.proxy.MethodFilter</td>
<td>12,143</td>
</tr>
<tr>
<td>7</td>
<td>org.apache.commons.dbcp.BasicDataSource</td>
<td>11,558</td>
</tr>
<tr>
<td>8</td>
<td>java.util.activation.DataHandler</td>
<td>11,176</td>
</tr>
<tr>
<td>9</td>
<td>org.apache</td>
<td>10,563</td>
</tr>
<tr>
<td>10</td>
<td>java.util.persistance</td>
<td>9,905</td>
</tr>
<tr>
<td>11</td>
<td>javax</td>
<td>9,527</td>
</tr>
<tr>
<td>12</td>
<td>spring</td>
<td>9,290</td>
</tr>
<tr>
<td>13</td>
<td>java.util.jws.WebService</td>
<td>9,258</td>
</tr>
<tr>
<td>14</td>
<td>org.apache.commons.lang.StringUtils</td>
<td>9,142</td>
</tr>
<tr>
<td>15</td>
<td>javax.xml.rpc.ServiceException</td>
<td>9,089</td>
</tr>
<tr>
<td>16</td>
<td>org.apache.objectweb.asm.Type</td>
<td>9,142</td>
</tr>
<tr>
<td>17</td>
<td>java.xml.namespace.QName</td>
<td>9,218</td>
</tr>
<tr>
<td>18</td>
<td>hibernate</td>
<td>7,922</td>
</tr>
<tr>
<td>19</td>
<td>java.util.xml.soap.SOAPException</td>
<td>7,381</td>
</tr>
<tr>
<td>20</td>
<td>javax.servlet.HttpServlet</td>
<td>7,724</td>
</tr>
<tr>
<td>21</td>
<td>xml</td>
<td>7,589</td>
</tr>
<tr>
<td>22</td>
<td>log4j</td>
<td>7,530</td>
</tr>
<tr>
<td>23</td>
<td>org.apache.commons.codec.DecoderException</td>
<td>7,289</td>
</tr>
<tr>
<td>24</td>
<td>net.sf.cglib.proxy.CallbackFilter</td>
<td>7,082</td>
</tr>
<tr>
<td>25</td>
<td>jstl</td>
<td>7,008</td>
</tr>
<tr>
<td>26</td>
<td>edu.emory.mathcs.backport.java.util.concurrent.ConcurrentHashMap</td>
<td>6,926</td>
</tr>
<tr>
<td>27</td>
<td>mail</td>
<td>6,686</td>
</tr>
<tr>
<td>28</td>
<td>servlet</td>
<td>6,664</td>
</tr>
<tr>
<td>29</td>
<td>com.sun.faces.config.ConfigureListener</td>
<td>6,516</td>
</tr>
<tr>
<td>30</td>
<td>org.apache.commons</td>
<td>5,838</td>
</tr>
<tr>
<td>31</td>
<td>javax.xml.stream.XMLStreamReader</td>
<td>5,838</td>
</tr>
<tr>
<td>32</td>
<td>javax.transaction.Synchronization</td>
<td>5,838</td>
</tr>
<tr>
<td>33</td>
<td>org.apache.commons.Log4j.Logger</td>
<td>6,311</td>
</tr>
<tr>
<td>34</td>
<td>j2ee</td>
<td>6,088</td>
</tr>
<tr>
<td>35</td>
<td>org.dom4j.DocumentException</td>
<td>6,036</td>
</tr>
<tr>
<td>36</td>
<td>org.apache.commons.discovery.tools.DiscoverSingleton</td>
<td>6,009</td>
</tr>
<tr>
<td>37</td>
<td>org.apache.commons.pool.impl.ObjectPool</td>
<td>6,500</td>
</tr>
<tr>
<td>38</td>
<td>javax.servlet.http.HttpServletRequest</td>
<td>5,886</td>
</tr>
<tr>
<td>39</td>
<td>axis</td>
<td>5,861</td>
</tr>
<tr>
<td>40</td>
<td>commons-logging</td>
<td>5,838</td>
</tr>
</tbody>
</table>

Table 3.1: Top 40 search queries from FindJar.com.
Download statistics for HTMLUnit show it has been retrieved 69,343 times for the versions we studied. Classes from within log4j appear as the 22nd and 33rd most popular search terms at FindJar.com for a total of 13,841 search queries. JDOM, key to the motivational scenario, continues to be heavily used at Chartwell Technology; FindJar.com has logged 3,425 search queries against it. For each system, we obtained as many versions as we could locate to examine the API changes between successive versions. Dig and Johnson [2006] constrained their study to “changes between major versions” and Xing and Stroulia [2006] limited their study to a single system and excluded “75% of all changes”. We included all changes between versions—both major and minor—in our analysis. With the exception of HTMLUnit, we excluded from consideration release candidate versions which often precede the official release, alpha releases not intended for widespread distribution (with one exception for log4j1.3-Alpha1.jar) and releases for which source code was not available. We feel that limiting ourselves to official releases is a small concession that still accurately captures the habits of industrial developers.

3.2 Study Design

To detect and analyze API changes we relied on a combination of automated and manual techniques. Two open-source tools were used for the automated portion of the study: the Eclipse API Tooling and JDiff. The former tool compares the compiled JAR file(s) of two library versions for binary incompatibility and generates a coarse-grained report of any differences found. The JDiff tool generates Javadoc from the library source code and then does a textual comparison of Javadoc from successive library versions to create a fine-grained report of any discrepancies.

These tools were used in conjunction with manual investigation as part of a three-step process: (1) we used the Eclipse API Tooling to determine the binary incompatibilities between successive versions; (2) we then used JDiff in an attempt to automatically classify the
API changes; and finally (3) we manually inspected and revised each of the reported changes (including looking through the associated documentation) to overcome shortcomings of these tools. To detect and analyze deprecation we utilized a custom Java Doclet to record usages of the \texttt{@deprecated} Javadoc tag and any supporting details such as the deprecation message or use of \texttt{@see} or \texttt{@link} tags.

### 3.3 Predicting API Change

Ideally API change should be limited and well documented so developers can plan ahead for migration. In practice we found that APIs change unpredictably and sometimes severely, and that API change is far from uncommon. There were binary incompatibilities in 87\% (48/55) of the successive library versions we tested. At times, the changes were surprisingly severe; between HTMLUnit 1.3 and 1.4 for example there were a “mere” 40 breaking changes but between versions 1.4 and 1.5 there were 298 breaking changes.

The severity of the changes might be tempered if developers could predict when they were going to occur. However, we examined several possible prediction criteria with little success. A change in the JAR version number is the most obvious indicator that a library has been updated. For example the Eclipse version number standard\(^1\) states that version numbers should be formatted as a 3-tuple of major#.minor#.service#. The major number is updated for breaking changes to the API and the minor number for “binary compatible API changes, significant performance changes, major code rework, etc.” In practice this rule was adhered to only 13\% of the time in the systems we studied (note that Eclipse itself was not among these).

We further hypothesized that developers might avoid API change by forgoing early versions and waiting for the library to mature and stabilize. For JDOM this hypothesis appears to hold true as 99.3\% (290/292) of the changes occur prior to the 1.0 release. Conversely for log4j

the hypothesis does not seem to hold; after a lull of five releases (1.2.11 to 1.2.15) there was a
spike of 197 changes in version 1.3.01. Similarly for HTMLUnit the quantity of changes after
the 2.0 release represents 43.8% (619/1413) of all changes. The lack of a consistent correlation
between library age and API stability also makes it a poor prediction tool.

Finally, we looked at the use of Deprecation as a mechanism for alerting developers to
impending API changes. The Javadoc authoring standards\(^2\) require that deprecation should be
accompanied by a “description in the first sentence [that] should at least tell the user when
the API was deprecated and what to use as a replacement …[furthermore the] @see tag (for
Javadoc 1.1) or @link tag (for Javadoc 1.2 or later) should be included that points to the
replacement method”. Accordingly the use of these ancillary tags is also examined. Lastly
Dig and Johnson [2006] state that “most API changes follow a long deprecate-replace-remove
[DRR] cycle to preserve backwards comparability” yet they offer no evidence as to the duration
of this cycle or the number of changes that adhere to it. Without such evidence it is not clear
that deprecation is being used as intended.

In our investigation, it quickly became apparent that deprecation fails in its role as a
harbinger of API change. Admittedly deprecation cannot be used for breaking changes that
involve the addition of new entities (e.g., adding a public method to an interface) but even after
accounting for such cases the total number of deprecation-compatible breaking changes in all
libraries exceeds by 33.4% (2013/1508) the number of entities listed as deprecated at any point
their collective history. The history of HTMLUnit contains some notable examples of this
problem: version 2.4 shows 54 deprecated entities yet version 2.5 contains 385 deprecation-
compatible breaking changes. The situation is repeated with HTMLUnit version 1.11 which
shows 8 uses of deprecation but version 1.12 has 91 binary incompatibilities with the previous
release.

Insufficient notice of all API changes might be tolerable if the changes for which warn-
\(^2\)http://java.sun.com/j2se/javadoc/writingdoccomments/#@deprecated [last accessed 28 April 2010].
nings were given could be counted on to occur. Unfortunately the results show that when the list of deprecated entities is not understating the volume of API changes it tends to overstate them instead. Version 1.2.3 of log4j lists 31 deprecated entities, yet in version 1.2.4 zero breaking changes occur. In fact the deprecation count for log4j increases almost continuously from version 1.2.1 (31 deprecations) to version 1.2.14 (53 deprecations) with very few of the entities actually moving beyond deprecation to the “remove” stage of the DRR cycle. Java itself contains some particularly egregious examples of entities that have been deprecated and for which replacements exist but removal has not occurred. For example java.awt.List.allowsMultipleSelections() was deprecated prior to Java 1.1 but is still present in Java 1.6—a period of time that spans almost 10 years!

Thus deprecation as a means for easing library evolution is limited from two perspectives. When deprecation is underutilized, breaking changes happen without warning and the DRR cycle is truncated to simply “replace and remove”. In this situation developers are forced to deal with changes as they happen. When deprecation is over-utilized it can lead to a DRR cycle that spans multiple releases and many years. In this case, developers could be forgiven for simply ignoring deprecation warnings since the changes often do not come to pass.³

As noted earlier @deprecation tags are supposed to be used in tandem with @see and @link tags to provide developers with suggestions for API replacements. Since few changes are properly documented in the first place it is not surprising that usage frequency of these additional tags is equally sporadic. For HTMLUnit only 49.9% (383/767) of deprecation tags were fully documented. For JDOM this value was 12.2% (11/99) and for log4j it was 39.1% (255/651).

On occasion, library designers did provide textual information about an API’s replacement. In the Java library this was often in the form of “…this has been replaced by…” statements.

³This problem is alluded to in the Javadoc standards guide for @deprecated (http://java.sun.com/j2se/1.4.2/docs/guide/misc/deprecation/deprecated.html [last accessed 28 April 2010]) which states, “It is probably not good to specifically mention a timetable for phase-out of the deprecated API [within the deprecation comments]; this is a business decision that needs to be communicated other ways.”
The designers of log4j and HTMLUnit also followed this textual approach of documenting changes. Such comments still represent a departure from the ideal since they (1) still rely on the developer to read comments and (2) cannot be easily parsed for the purposes of tool support.

Overall, we argue that if the library providers do not even point out how to cope with version transitions, they are unlikely to create heavyweight specifications that could be used to correctly transform client code, as required by a variety of API migration research [Chow and Notkin, 1994, Tip et al., 2003, Balaban et al., 2005, Nita and Notkin, 2010]. A developer is left with little choice but to manually transition their code, a notably painful process [Boshernitsan and Graham, 2004]. More pragmatic support would be a boon.

3.4 Study Results

Our in-depth study of API changes is reported at two different levels of granularity. First we examined the results of running Eclipse API Tooling in isolation to see what types of binary incompatible changes occur in general. For example, API Tooling reports a method as broken if (1) its name changes; (2) parameters for it are added, deleted or reordered; (3) its return type or exceptions thrown change; or (4) it has legitimately been removed. Despite the absence of detailed information such information was useful in determining the kinds of entities most susceptible to change. Second, we used JDiff and manual inspections of the documentation (both release notes and Javadoc) to classify API changes at a more detailed level. This additional step allowed us to distinguish cases where a method was “broken” because of parameter reordering, from cases arising because of some other modification to the method signature.

Prior to presenting the API study results there are some important caveats to consider. A single binary incompatibility may conceal multiple API changes. For example a method may be reported as removed when in fact it was renamed and an additional parameter was added.
In the detailed API analysis this single incompatibility is reported as three separate changes.

3.4.1 High-Level Results

Modifications to methods represented the largest number of binary compatibility changes with binary method removal—which in this context may be a change to the signature or legitimate removal—being the most common. For HTMLUnit, binary method removals represent 65.6% (928/1413) of all changes and the values for JDOM and log4j were 64.0% (187/292) and 46.6% (191/414) respectively. This corresponds well with the breaking changes reported by Dig and Johnson [2006] where method-related modifications comprised 53.8% (70/130) to 64.5% (129/200) in the two large open-source systems they studied: Eclipse and Struts\(^4\). Xing and Stroulia [2006] examined entity deletion, moves, and renaming within Eclipse and showed that methods account for the majority of moves, at 59.1% (1369/2315), and renamed entities, at 76% (3512/4582). Thus, both bodies of work support the notion that methods (as opposed to packages, classes, etc.) are the source code entities most prone to change.\(^5\)

Like Xing and Stroulia [2006], we found few changes in entity visibility. Of the total number of changes, visibility reduction accounted for 2.6% (11/414), 14.0% (41/292) and 0.005% (8/1413) in log4j, JDOM and HTMLUnit. Similar to Xing and Stroulia [2006], we found that a the changes were predominately in a handful of entities; in log4j 81.1% (9/11) of the changes were in a single class, in JDOM 60.9% (25/41) were in three classes and in HTMLUnit 100% (8/8) of visibility reductions were due to one class.

Similarly we found—as Xing and Stroulia [2006] did with Eclipse—that the class hierarchy of each of the studied systems was relative stable. Binary incompatibility arising from a reduction in the superclass or superinterface hierarchy were relatively rare events. Of all changes, only 1.2%, 1.3% were due to hierarchy changes in log4j and JDOM. In HTMLUnit, up until version 2.4 such changes accounted for 1.26% (13/1026) of all changes; in the final version


\(^{5}\)The authors provide no per-entity breakdown of data on entity removal.
2.5, there is a spike of 135 such changes which is 34.8% (135/387) of the total changes in that release. This final change surge may be an outlier as the 2.5 release marks the beginning of an extensive rewrite of the HTMLUnit library.

3.4.2 Low-Level Changes

All the changes reported by the API tooling had the potential to break existing clients. Our goals during in-depth investigation of these changes were (1) to discover more details for binary incompatibility changes that were too broad and (2) to analyze these details to determine which ones might be easily corrected if support for refactoring of references was possible.

In their respective studies, Dig and Johnson [2006] as well as Xing and Stroulia [2006] focused on analyzing changes to determine if those changes could have arisen from library designers using refactorings. The former study looked at library source code and documentation to make an expert determination of whether or not a given change was the result of a refactoring while the latter study used an inference algorithm [Xing and Stroulia, 2005a] for the same purpose. In both studies, the emphasis is on reconstructing the actions of the library designer with the intention of repeating those actions for the library user. Accordingly Dig and Johnson [2006] “propose that tools that record-and-playback refactorings are used to update applications”. Similarly Xing and Stroulia [2006] argue that “client applications can be automatically updated by a refactoring migration tool if the relevant information of the refactored components can be gathered through the refactoring engine”.

Conversely, work by other researchers [Murphy-Hill et al., 2009, Murphy et al., 2006] shows that (a) many refactorings are performed without the help of refactoring tools, (b) refactoring tools are underused in general (though some simple refactorings like rename are common), and (c) “toolsmiths” (analogous to library designers in our work) are themselves more likely to forgo tools and refactor manually. Consequently we did not attempt to categorize changes in terms of the refactoring that might have been applied, although we did find, consis-
tent with Dig and Johnson [2006] and Xing and Stroulia [2006], that a large number of changes were attributable to renameing, we do not quantify this in our findings. In this respect, the other two studies of API change can be considered more detailed than ours. Instead we provide a qualitative report of some of the more interesting cases of API change (1) that are not readily handled by lexical/syntactic tools and (2) for which refactoring support is currently limited or absent.

Net Effect of Refactoring
In some cases multiple refactorings by library designers manifested themselves as a single refactoring that had to be carried out by library users. A review of the Chartwell library migration scenario (see Table 2.1) helps illustrate this distinction. Consider the change from `Element.getParent()` to `Element.getParentElement()`. From the perspective of the library designer this change involved the creation of a new class (`Content`), a change in the inheritance hierarchy of `Element` to introduce `Content` as a superclass, followed by a “pull up method” refactoring to move `getParent()` into the superclass and then renaming it. From the perspective of the library user, all of this simply appears to be a “method rename refactoring” wherein `getParent()` is replaced with `getParentElement()`.

Inconsistent Changes
In the transition from JDOM-b7 to JDOM-b8 an `XMLOutputter` class was introduced to provide a string-based XML representation of various XML entities suitable for output to the console. This was to replace the numerous separate `getSerializedForm()` methods that were declared on the `Attribute`, `CDATA`, `Comment`, `DocType`, `Document`, `Element`, and `ProcessingInstruction` classes. An overloaded `XMLOutputter.outputString(...)` method was provided for all the classes `Attribute`. Thus, method invocations of the form `obj.getSerializedForm()` could be replaced with `(new XMLOutputter()).outputString(obj)` except in cases where `obj` was an object of the
type attribute. Making this type of inconsistent change requires a semantic understanding of the code.

**Replacing Method Expressions**

The `Element.getCopy(String)` method in JDOM-b7 was deprecated with the comment: “use clone() and setName() instead”. Assuming the original statement was `e1.getCopy(str)` (where `e1` is of type `Element`) then the revision could be expressed as the following sequence: `Element e2 = e1.clone(); e2.setName(str);`. Alternatively, if we allow the replacement of method expressions then the compound statement can be written in a single line: `((Element) e1.clone()).setName(str);`. Other researchers [Henkel and Diwan, 2005, Balaban et al., 2005] have argued that the ability to replace method expression can be useful.

**Introduce Parameter Object**

Building upon the previous example, there are situations where replacing parts of the method argument list with an expression can be useful. The JDOM method `Element.removeAttribute(String name, String uri)` is removed in favour of a method where the `uri` string is enclosed within a larger `Namespace` object. Conveniently, the `Namespace` object exposes a static factory method that accepts strings and returns a `Namespace` so method invocations can be refactored to: `Element.removeAttribute(name, Namespace.getNamespace(uri));`. This is arguably an “Introduce Parameter Object” refactoring since one or more method parameters are replaced by a single object; it should also be noted that this refactoring currently lacks support within the Eclipse IDE.

**Responding to Deprecation**

Consider what needs to happen from the library user’s perspective even if a library follows the DRR cycle meticulously. Let us revisit the change of `setUpdated(...)` to `setTime-
The library designers first deprecate the original method and introduce the replacement, allowing both to co-exist for six months before finally moving to removal. What must a developer do in response? Even while within the deprecate-replace transition phase, he has no option but to manually locate and rename every `setUpdated()` call to `setTimeStamp()`. The deprecation warning is informative but it provides no assistance in actually making the necessary change. If, however, the developer were able to transform references directly, he could select all the invocations of `setUpdated()` and rename as required. It is worth noting that in this scenario we have full binding information, yet still the ability to transform references separately from declarations can be useful.

Table 3.2 summarizes the results over all transitions between successive versions. Tables 3.3, 3.4, and 3.5 present the detailed results of the binary incompatibility analyses over multiple version transitions for HTMLUnit, JDOM, and log4j respectively.

Tables 3.6, 3.7, and 3.8 present the counts of the number of deprecated entities and total number of binary incompatibilities detected. In cases where a binary incompatibility was not or could not be further categorized, there will be a discrepancy with the detailed results tables.

### 3.5 Summary

We found that APIs in these systems change unpredictably and sometimes severely, and that API change is far from uncommon. In addition to the data, we also found that the @depre\-cated tag is an unreliable guide: deprecated entities do not always get eliminated, moreover entities can be deleted or otherwise transformed without ever having been labelled as deprecated and without any explicit indication as to how a developer ought to migrate their library usage to a newer version.

As the average number of changes for each transition between versions is between 13 and 28, the burden on the developer to determine how to remap each dangling reference is
Table 3.2: API breakages in the studied systems. Mean and standard deviation are presented for the inter-version binary incompatibilities both with respect to all changes and broken down by entity kind.

4.2 Application and Features

Trident is applied in a 5-stage process. (1) The developer highlights code to refactor and activates Trident (selection). (2) A wizard is displayed allowing for specific details about the search criteria to be configured (search configuration). (3) A checkbox list is presented with each location described that matched the criteria; by default, the entire set of results is selected, but this can be restricted to any individual locations (restriction). (4) Another wizard is displayed allowing for specific details about the refactoring criteria to be specified (refactoring configuration). (5) A comparison editor allows for “before” and “after” shots of the changes to be previewed. At any step, the process can be aborted, or the developer can return to the previous step. We describe each stage of the process, below, with the use of a running example: the statement Category.getInstance(ResourceBundleTest.class), in which Category and ResourceBundleTest are type references.

Selection...

The Selection stage begins with the developer highlighting a section of code containing a reference in the editor; he activates Trident using a button on the Eclipse toolbar. Trident then obtains the node (as defined by the Eclipse Java Development Tools) from the abstract syntax tree (AST) corresponding to the current selection; to determine likely resolution information in the presence of dangling references, the partial program analysis (PPA) tool of Dagenais and Hendren [Dagenais and Hendren(2008)] is applied. Currently only 6 types of AST node selections are supported: Simple Name, Simple Type, Qualified Name, Qualified Type, Method Invocation, and Class Instance Creation.

Search Configuration...

The developer specifies how the exemplar should be used to search for other dangling references. The developer is asked to identify which portions of the exemplar should be included in the search and how they should be interpreted. Developers are provided with (a maximum of) three search options to guide the search. Figure 1 illustrates this with our running example. The method invocation is broke into its three constituent components: method expression (Category), method name (getInstance), and a variable length argument list (ResourceBundleTest.class). Beside each component is a drop down box with search options. For the method expression three search options are available: “verbatim”, meaning search for method expressions that are lexically identical to Category; “type”, meaning search for method expressions that evaluate to the same type, which in this case is org.apache.log4j.Category; and “ignore”, meaning remove that portion of the exemplar from search consideration. Method names have only two search options: “verbatim” and “ignore”. As previously, choosing verbatim in our example means a search will be conducted for other method invocations where the
Table 3.3: Binary incompatibilities between versions of HTMLUnit. "∆" denotes a change, and "∆↓" denotes more specifically a reduction.

<table>
<thead>
<tr>
<th>Field</th>
<th>Version</th>
<th>Types</th>
<th>Methods</th>
<th>Constructors</th>
<th>Fields</th>
<th>Functions</th>
<th>Visibility</th>
<th>Parameter Type</th>
<th>Parameter Type</th>
<th>Result Type</th>
<th>Parameter Type</th>
<th>Parameter Type</th>
<th>Parameter Type</th>
<th>Parameter Type</th>
<th>Parameter Type</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0pre1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0pre2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0pre3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0pre4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0pre5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1pre1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1pre2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1pre3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1pre4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.1pre5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: "∆" denotes a change, and "∆↓" denotes more specifically a reduction.
<table>
<thead>
<tr>
<th>Fields</th>
<th>Constructors</th>
<th>Methods</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 3.4: Binary incompatibilities between versions of JDOM. "∆" denotes a change, and "∆↓" denotes more specifically a reduction.
<table>
<thead>
<tr>
<th>Fields</th>
<th>Constructors</th>
<th>Methods</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant value $\Delta$</td>
<td>visibility $\Delta$</td>
<td>deleted</td>
<td>+Final</td>
</tr>
<tr>
<td>parameter moved</td>
<td>parameter</td>
<td>visibility $\Delta$</td>
<td>deleted</td>
</tr>
<tr>
<td>exception moved</td>
<td>parameter moved</td>
<td>visibility $\Delta$</td>
<td>deleted</td>
</tr>
<tr>
<td>parameter type $\Delta$</td>
<td>parameter moved</td>
<td>visibility + $\Delta$</td>
<td>push up method</td>
</tr>
<tr>
<td>parameter type $\Delta$</td>
<td>parameter</td>
<td>visibility + $\Delta$</td>
<td>pull down method</td>
</tr>
<tr>
<td>method type $\Delta$</td>
<td>visibility $\Delta$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>class $+$ interface $\Delta$</td>
<td>visibility $\Delta$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3.5: Binary incompatibilities between versions of log4j. $\Delta$ denotes a change, and $\Delta \downarrow$ denotes more specifically a reduction.
<table>
<thead>
<tr>
<th>Version transition</th>
<th>Deprecations</th>
<th>Binary incompatibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0pre1–1.0pre2</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1.0pre2–1.0pre3</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1.0pre3–1.0pre4</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>1.0pre4–1.0pre5</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>1.0pre5–1.0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>1.0–1.1pre1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>1.1pre1–1.1pre2</td>
<td>2</td>
<td>17</td>
</tr>
<tr>
<td>1.1pre3–1.1pre4</td>
<td>7</td>
<td>19</td>
</tr>
<tr>
<td>1.1pre4–1.1pre5</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1.1pre5–1.1rc1</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1.1rc1–1.1rc2</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>1.1rc2–1.1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1.1–1.2</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>1.2–1.2.1</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>1.2.1–1.2.2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1.2.2–1.2.3</td>
<td>6</td>
<td>19</td>
</tr>
<tr>
<td>1.2.3–1.3pre1</td>
<td>6</td>
<td>54</td>
</tr>
<tr>
<td>1.3pre1–1.3pre2</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>1.3pre2–1.3</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>1.3–1.4</td>
<td>9</td>
<td>40</td>
</tr>
<tr>
<td>1.4–1.5</td>
<td>17</td>
<td>29</td>
</tr>
<tr>
<td>1.5–1.6</td>
<td>25</td>
<td>298</td>
</tr>
<tr>
<td>1.6–1.7</td>
<td>26</td>
<td>14</td>
</tr>
<tr>
<td>1.7–1.8</td>
<td>5</td>
<td>42</td>
</tr>
<tr>
<td>1.8–1.9</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1.9–1.10</td>
<td>5</td>
<td>14</td>
</tr>
<tr>
<td>1.10–1.11</td>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>1.11–1.12</td>
<td>100</td>
<td>93</td>
</tr>
<tr>
<td>1.12–1.13</td>
<td>116</td>
<td>7</td>
</tr>
<tr>
<td>1.13–1.14</td>
<td>130</td>
<td>126</td>
</tr>
<tr>
<td>1.14–2.0</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>2.0–2.1</td>
<td>27</td>
<td>6</td>
</tr>
<tr>
<td>2.1–2.2</td>
<td>44</td>
<td>97</td>
</tr>
<tr>
<td>2.2–2.3</td>
<td>46</td>
<td>17</td>
</tr>
<tr>
<td>2.3–2.4</td>
<td>51</td>
<td>112</td>
</tr>
<tr>
<td>2.4–2.5</td>
<td>41</td>
<td>387</td>
</tr>
</tbody>
</table>

Table 3.6: Raw counts of deprecated entities and total binary incompatibilities between versions of HTMLUnit.
<table>
<thead>
<tr>
<th>Version transition</th>
<th>Deprecations</th>
<th>Binary incompatibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>b3–b6</td>
<td>3</td>
<td>73</td>
</tr>
<tr>
<td>b6–b7</td>
<td>30</td>
<td>25</td>
</tr>
<tr>
<td>b7–b8</td>
<td>10</td>
<td>47</td>
</tr>
<tr>
<td>b8–b9</td>
<td>17</td>
<td>25</td>
</tr>
<tr>
<td>b9–b10</td>
<td>30</td>
<td>120</td>
</tr>
<tr>
<td>b10–1.0</td>
<td>0</td>
<td>n/a</td>
</tr>
<tr>
<td>1.0–1.1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1.1–1.1.1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.7: Raw counts of deprecated entities and total binary incompatibilities between versions of JDOM.

<table>
<thead>
<tr>
<th>Version transition</th>
<th>Deprecations</th>
<th>Binary incompatibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0.4–1.1.3</td>
<td>100</td>
<td>36</td>
</tr>
<tr>
<td>1.1.3–1.2.1</td>
<td>31</td>
<td>155</td>
</tr>
<tr>
<td>1.2.1–1.2.2</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>1.2.2–1.2.3</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>1.2.3–1.2.4</td>
<td>31</td>
<td>0</td>
</tr>
<tr>
<td>1.2.4–1.2.5</td>
<td>31</td>
<td>5</td>
</tr>
<tr>
<td>1.2.5–1.2.6</td>
<td>33</td>
<td>3</td>
</tr>
<tr>
<td>1.2.6–1.2.7</td>
<td>37</td>
<td>n/a</td>
</tr>
<tr>
<td>1.2.7–1.2.8</td>
<td>37</td>
<td>7</td>
</tr>
<tr>
<td>1.2.8–1.2.9</td>
<td>49</td>
<td>9</td>
</tr>
<tr>
<td>1.2.9–1.2.11</td>
<td>49</td>
<td>0</td>
</tr>
<tr>
<td>1.2.11–1.2.12</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>1.2.12–1.2.13</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>1.2.13–1.2.14</td>
<td>53</td>
<td>0</td>
</tr>
<tr>
<td>1.2.14–1.2.15</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>1.2.15–1.3.01</td>
<td>35</td>
<td>197</td>
</tr>
</tbody>
</table>

Table 3.8: Raw counts of deprecated entities and total binary incompatibilities between versions of log4j.
potentially high—especially since a single API breakage could result in multiple dangling references. We note that the majority of change events (~62%) involve the deletion of entities, and approximately half of the change events are specifically method deletions. (We do not count as method deletions any likely signature changes that would be easily located by the developer.) Deletions are potentially the greatest burden to the developer, as no immediate clues exist about what the deleted entity should be replaced with. We found that most of these changes were not accompanied with documentation about the recommended migration path.
Refactoring References: Prototype Tooling

As a first step in determining whether tool support for refactoring references would be of practical benefit, we present our prototype tool, Trident. Trident is implemented as a plugin for the Eclipse IDE, which aims to provide flexible search-and-replace functionality for refactoring references. Trident does not currently aim to address every detail of the process of refactoring references: by investing in partial support, we intended to collect enough empirical evidence to inform whether stronger tooling is worth developing.

We wished to mimic the capabilities of existing Eclipse tools (such as Java Search, and Eclipse’s refactoring support), but allow for intelligent replacements in contexts where existing support could not operate: incomplete code bases with dangling references.

Section 4.1 describes the ideas behind our approach, which fed into the design and construction of Trident; these are then solidified into goals in Section 4.2. Section 4.3 describes how Trident is used by a developer to refactor references. Section 4.4 provides salient details of the algorithms and implementation of Trident.

4.1 A Consideration of the Required Features

We explain the current design of Trident by revisiting some of the design deliberations that contributed to its development. To this end, we recap the `Element.getParent()` to `Element.getParentElement()` API change from the motivating example.

Figure 4.1 captures the state of the code immediately after a Chartwell developer has replaced the JDOM-b9 JAR file with JDOM-b10. Confronted with this screen, the Chartwell developer searches the new Javadoc to find that calls to `getParent()` should be replaced...
Figure 4.1: Screenshot of `getParent()` dangling reference.

```java
Element header = rsp.getParent().getChild("msg");
```

with `getParentElement()`—with the exception, as previously pointed out, of invocations of `getParent()` that occur on variables of type `Attribute`. Now consider how a developer might describe the corrective transformation that needs to occur: “Find every invocation of `getParent()` that is being called on an object that is of the same type as the variable `rsp`, rename it to `getParentElement()`, and retain any subsequent method calls (like `getChild()`) that might appear later in the expression”. Other researchers have found that developers describe changes in terms of high-level programming concepts, and when renaming entities, they reason in terms of the type and scope of the variables involved [Boshernitsan et al., 2007, Sillito et al., 2008].

Keeping this hypothetical description in mind, let us review the tool support options available to the developer. (1) He could perform a lexical search-and-replace for the literal string "getParent". What this solution offers in terms of simplicity, it offsets in terms of side effects. Search-and-replace will correct all the method invocations but it will also alter any literal strings declared within the code that contain the substring `getParent`, any Javadocs or comments related or unrelated to this change that share the same string as well as any other methods whose name happens to contain the substring `getParent`. (2) Leveraging the abstract syntax tree (AST), he could specify that only method invocation tree nodes should be transformed, which eliminates any accidental changes to literal strings or comments. However he is still left with the problem of other methods that share the same name but are not invoked on variables of the same type. There are other limitations to lexical and syntactic techniques that are mentioned in Sections 1 and 5.

To satisfy our putative developer, what we would like to be able to do is simply select the
dangling reference, rename it, and leave all the other details of the transformation for some software tool to figure out—with user prompts in the event of errors or ambiguity—just as we would in a standard “Rename” refactoring. Earlier work in the area of tool support for API evolution has attempted to realize this hypothetical objective in various ways.

CatchUp! by Henkel and Diwan [2005] records refactoring actions taken by library designers during the creation of a new library version and then allows library users to play back these actions on their own source code thereby “catching up” to the latest changes. The idea behind CatchUp! is certainly appealing; if in fact the designers of JDOM used a “Rename” refactoring to change `getParent()` to `getParentElement()`, our fictional developer would be presented with renaming functionality that meets his needs of handling details such as verifying the type of the `rsp` variable and requires minimal developer intervention.

The notion that refactoring of declarations could be made temporally independent from the refactoring of references certainly contributed to Trident’s design. However CatchUp! as it stands has numerous limitations that make it unsuitable for our purposes. CatchUp! places the burden of responsibility on library designers to record and ship change information which, judging by the poor deprecations and documentation practices of designers discussed in Section 3, they are disinclined to do. Library designers and developers must also use the same IDE, as refactorings recorded in one IDE cannot be replayed in another. Most importantly CatchUp! still relies on the availability of both declarations and references to generate the bindings necessary to carry out its functions. To satisfy these needs, “CatchUp! generates Java Source code stubs . . . [that] provide declarations for the classes, methods and fields that are part of the API” [Henkel and Diwan, 2005, p. 280].

This solution may suffice in some limited cases of library migration scenarios, but it differs from the more general problem of dangling references that we set out to consider. As noted in Section 1 there are several areas where refactoring of references is of potential benefit.

(1) In test-driven development scenarios unit tests are created before the entities they are in-
tended to test exist. (2) In pragmatic reuse scenarios where developers copy code from outside their system and must adapt the copied references to use declarations within their own system. (3) Practical scenarios arise where developers wish to alter references without altering declarations. For instance, the set of logging-related methods in an application might be subdivided such that half the references need to be remapped to use the log4j framework and the other half continue to use the internal implementation. This is similar to the situation that Henkel and Diwan envisioned when they advocated for “a ‘Remove Method’ refactoring . . . which allows users to replace all calls with a [different] Java expression” [Henkel and Diwan, 2005, p. 282]. Despite these distinctions, the overall concept that refactorings could be leveraged to alter references owes some of its origin to CatchUp!

Work by Balaban et al. [2005] also proposed a mechanism for updating client code to use new libraries which relied on the idea of migration specifications. In this approach, the library user, not the library designer, writes out a series of rules that describe how usages of the old API should be rewritten to conform to the new API. For example, \( \text{new Hashtable()} \rightarrow \text{new HashMap()} \) is a rule in the migration specification for moving from the legacy Hash-Table class to the new Java collection framework. The migration specification also allowed developers to operate outside the strict limits of standard refactorings by permitting changes to method expressions and reusing method parameter names. One of the migration rules allowed instances of the legacy \( \text{java.util.Vector} \) class to be removed with the aid of a user-designed auxiliary called \( \text{Util} \) as follows:

\[
\text{myVector.indexOf}(\text{obj}, 1) \rightarrow \text{Util.indexOf(myVector, \text{obj}, 1)}
\]

A closer read of this rule reveals some novel functionality: the variable name \( \text{myVector} \) originally used as the method expression has been replaced with the expression \( \text{Util} \). In Balaban’s example the \( \text{Util} \) class exposes only static methods so the creation of an additional instance variable is not required. However, the idea that a method expression should also be
replaceable is a break from tradition and, in actual API migration, this situation does occur. In fact, allowing the method name to be replaced with an arbitrary expression—not just a new name—can also be useful. In Java version 1.4.2 there are three instances where a Deprecated method name could be removed if there were tool support for changing `disabled()` to `setEnabled(false)`.

The second item to note is that the migration specification allows for parameterized replacements. The `myVector` variable on the left hand side of the rule is reused as the name of a method parameter on the right hand side. Much like the backreference feature of regular expressions, this allows matched terms to be reused in the search replacement.

Some of these ideas are incorporated into Trident’s design, but Balaban’s approach has a number of shortcomings that curtail its applicability in our target contexts. Like CatchUp! the work relies on bindings to proceed, but the larger issue is the need to describe the API migration in its entirety using a formal specification language. There is an aversion among developers to learning and then crafting task descriptions in arbitrary formal formats [Boshernitsan and Graham, 2004]. Also, the complete specification has to be written in advance. Unless you know with certainty that your application uses a small fraction of the API calls—in which case you can just write migration rules for that subset—you need to account for all possible transformations from the old API to the new API. Figure 4.2 shows the full specification provided by Balaban et al. for migrating from `Vector` to `ArrayList`—a fairly trivial change—to give a sense of how lengthy such specifications can be.

The CatchUp! tool and the work of Balaban et al. yielded a number of ideas of what to do—and what not to do—that influenced the design of Trident. These are summarized as follows:

**Developer Independence.** Migration specifications allowed the developer to correct for API change without reliance on the library designer. Unlike with CatchUp!, no additional effort is required or expected from library designers. We sought to emulate this.
Figure 4.2: The full specification required by the method of Balaban et al. [2005] to migrate from Vector to ArrayList.
**Refactoring-Like Support.** We wanted developers to be able to correct dangling references using tools modeled after refactorings and avoid having the developer provide detailed formal descriptions of change tasks. Modeling refactoring also meant providing the preview, undo and iterative ability to transform code that traditional refactorings support.

**Refactoring-Unlike Support.** Henkel and Diwan [2005] suggested and Balaban et al. [2005] implemented the ability to replace a Java expression with another expression such as the change from `myVector` to `Util`. This also encompasses the ability to reuse terms from the original expression in the replacement expression.

Finally, we wanted to adhere as closely as possible to these design ideas while operating in the presence of broken semantics (i.e., no bindings) and possibly broken syntax.

### 4.2 Goals

When refactoring references, we wanted to provide support comparable in capabilities to existing refactoring tools, and not simply extend lexical/Java Search to include a replace feature. After spending some time observing Eclipse users’ invocations of the “Rename” and the “Introduce Local Variable” refactorings that Eclipse provides, we made several observations about the nature of refactoring which we sought to emulate:

**Exemplar Based.** To activate most refactorings the developer must provide an example of the code to be refactored. For instance, renaming a method requires that the cursor is currently placed on a method name.

**Completion Assistance.** Once you have selected the type of refactoring to apply, a specialized UI provides completion assistance. Explicit assistance in the “change method signature” refactoring comes in the form of type-completion widgets beside the method arguments that allow you to quickly select from other types currently visible in the project. Implicit
assistance comes in the form of checking for other methods in the same scope that might share your proposed method name/signature and cause naming conflicts.

**Escape Route.** A refactoring can be aborted at various stages and for various reasons. Most refactorings provide an inline preview of the proposed change so the user has immediate visual feedback on whether or not he should proceed. Following that, a preview is provided that lists all affected files and shows a side-by-side comparison (with syntax highlighting). Again, at this stage, the developer can cancel the refactoring altogether or selectively override the refactoring and exclude some files from the change. Even once the code has been modified it is possible to undo all the changes on a project wide basis.

4.3 Application and Features

Trident is applied in a 5-stage process. (1) The developer highlights code to refactor and activates Trident (*Selection*). (2) A wizard is displayed allowing for specific details about the search criteria to be configured (*Search Configuration*). (3) A checkbox list is presented with each location described that matched the criteria; by default, the entire set of results is selected, but this can be restricted to any individual locations (*Restriction*). (4) Another wizard is displayed allowing for specific details about the refactoring criteria to be specified (*Refactoring Configuration*). (5) A comparison editor allows for “before” and “after” snapshots of the changes to be previewed, and the developer can choose to proceed with the changes (*Preview and Enactment*). At any step, the process can be aborted, or the developer can return to the previous step. We describe each stage of the process, below, with the use of a running example: the statement `Category.getInstance(ResourceBundleTest.class)`, in which `Category` and `ResourceBundleTest` are type references.
Figure 4.3: Trident search configuration.

Figure 4.4: Trident search results.
Figure 4.5: Trident refactoring configuration.

Figure 4.6: Trident reference refactoring preview.
4.3.1 Selection

The Selection stage begins with the developer highlighting a section of code containing a reference in the editor; he activates Trident using a button on the Eclipse toolbar.

Trident then obtains the node (as defined by the Eclipse Java Development Tools) from the abstract syntax tree (AST) corresponding to the current selection; to determine likely resolution information in the presence of dangling references, the partial program analysis (PPA) tool of Dagenais and Hendren [2008] is applied. Currently only 6 types of AST node selections are supported: Simple Name, Simple Type, Qualified Name, Qualified Type, Method Invocation, and Class Instance Creation.

4.3.2 Search Configuration

The developer specifies how the exemplar should be used to search for other dangling references. The developer is asked to identify which portions of the exemplar should be included in the search and how they should be interpreted. Developers are provided with (a maximum of) three search options to guide the search.

Figure 4.3 illustrates this with our running example. The method invocation is broken into its three constituent components: method expression (Category), method name (getInstance), and a variable length argument list (ResourceBundleTest.class). Beside each component is a drop down box with search options. For the method expression, three search options are available: “verbatim”, meaning search for method expressions that are lexically identical to Category; “type”, meaning search for method expressions that evaluate to the same type, which in this case is org.apache.log4j.Category; and “ignore”, meaning remove that portion of the exemplar from search consideration. Method names have only two search options: “verbatim” and “ignore”. As previously, choosing verbatim in our example means a search will be conducted for other method invocations where the name portion is getInstance.
Arguments have the same three search options available with one small difference. If there are \( n \geq 2 \) arguments, choosing “ignore” for any one of them still requires the search condition that is applied to the others to hold and the number of arguments must still equal \( n \). Choosing to ignore all the arguments means any type and any number of arguments is considered valid in the ensuing search.

The search criteria are translated into lexical search criteria via regular expressions as well as additional semantic checks when appropriate.

4.3.3 Restriction

Locations that match the search criteria are presented in a checkbox list. The developer can review the individual matches, select/deselect these individually or as a group, or back up to revise their search criteria. In Figure 4.4, we see that multiple instances of calls to `Category.getInstance(...)` have been found, where the details of the argument vary between a variety of class literals and invocations of `getName()` on class literals (which returns a `String`). The developer has selected two of the locations as being of interest.

4.3.4 Refactoring Configuration

The developer can then specify how the components should be altered, as illustrated in Figure 4.5. This step is relatively simplistic in our current prototype. The wizard is initially populated with details from the exemplar. If a given component is unaltered by the developer, the corresponding component in all locations is left untransformed. Otherwise, the corresponding component in all locations is transformed as specified: (1) the method name can be replaced—if replaced via the “Browse for method” button, the specific method to be invoked will be identified and hence `import` statements can and will be modified automatically as well; (2) the method expression can be replaced; (3) the existing method arguments can be replaced; (4) any existing argument can be deleted; (5) new arguments can be inserted; and (6) arguments can be
moved, which does not transform the contents of corresponding arguments, only their positions in the list.

To be clear, this set of possible transformations is purposefully limited to simplify the process of using it. Through the Restriction stage and iterative invocations of Trident, we feel that the completion of a large task is more practicable than with complex specifications.

4.3.5 Preview and Enactment

As a final stage, the developer can preview the change that will result at each selected location, in a comparison editor that is standard for automated refactorings in Eclipse (as in Figure 4.6). If the previewed transformation is unacceptable at any point, the developer can deselect that location, thereby proceeding to refactor only the other locations, or he can back up to an earlier stage of the process to revise his criteria. Finally, the transformations are applied in a single, undoable step, so the developer can globally undo the transformation if unexpected and unacceptable problems arise.

4.4 Tool Implementation

In this section we give a detailed explanation of the internal workings of Trident’s five stage process. This section also highlights applications of some of the original design ideas.

4.4.1 Selection

Trident begins with the developer selecting an exemplar of the code requiring correction. Since methods (including constructors) are the part of an API most prone to change (see Section 3), our explanation (and development efforts) focus on that use case. Renaming of entities was also a common change so we support the transformation of type references (via SimpleType and QualifiedType AST nodes) and non-type references (via SimpleName and QualifiedName AST nodes). The examples below show how some common Java statements are
converted to the AST node classes that Trident operates on.

```java
public String name = new String("joe");
```

```java
public java.lang.String name = org.example.Name.JOE;
```

In our earlier prototypes, developers expressed frustration with selecting the correct code snippet for Trident to operate on. Some developers would select too small a section of code (e.g., `String`) while others would select too much (e.g., `String name`). To correct for this, Trident now includes a selection assistance feature. If a developer selection does not match an AST node that Trident supports, a dialog box offers to alter the selection to the “nearest” AST node within the same line that is supported. Figures 4.7 and 4.8 show before and after images of this.

Once a selection is made, the PPA tool [Dagenais and Hendren, 2008] is applied to the current Java file to recover as much resolved (i.e., binding) information as possible. For example, if the class `Category` in our running example had been removed in the latest library version, PPA would attempt to recover its fully qualified name (FQN). If there was an explicit `import` statement that ended with `Category` then that would be used to form the FQN. If there were one or more wildcard `import` statements preventing precise determination of the FQN it would remain unresolved.¹ In situations where the PPA algorithm fails to create bindings, a warning is provided to the developer indicating this fact and it is left up to the developer to decide if he wishes to proceed.

¹Full details of the inference strategies employed can be found in the original paper by Dagenais and Hendren [2008].
4.4.2 Search Configuration

The inclusion of a search screen is also an outcome of earlier prototype testing. Consider selecting the following statement (where `cat` is a variable of type `Category`): `cat.getInstance(ResourceBundleTest.class)`. Each developer could have a different interpretation of how this exemplar should be interpreted to locate other relevant method invocations. Some would want only calls to `getInstance(...)` made on variables named `cat`. Others would want all calls to `getInstance(...)` made on any variable that shared the same type as `cat`. While others would want any call to a method named `getInstance` regardless of the variable it was called on, but where the argument is of type `Class`. Rather than make assumptions on behalf of the developer, we decided to allow them to explicitly state their search criteria (as “verbatim”, “same type”, or “ignore” for each of the various parts of the statement).

The search implications of the “verbatim” and “ignore” options are relatively straightforward but the “same type” option hides some details. A method invocation has more than one “type” associated with it; there is the return type of the method and its declaring type. We ignore the former and concentrate on the latter. Yet, even when referring to the declaring type there can be ambiguity: should we differentiate between cases where the method is actually declared in a class or inherited from above? When using the “same type” option on `cat.getInstance(...), should we return only locations where `getInstance(...)` is called on variables of type `Category` or also locations where the method is called statically (i.e., as `Category.getInstance(...)`) or via the fully qualified class name (i.e., `org.apache.log4j.Category.getInstance(...)`)? The final decision was to treat method invocations on type variables, simple type names, and fully qualified type names as equivalent and ignore issues relating to inheritance. The kind of node selected in the previous screen is implicitly included as a criterion in the current search. For instance, if a method invocation is used as the exemplar, only method invocations will be examined during

--2The method `getInstance(Class)` is a static method
Once the developer has formulated a search query, a specialized AST visitor [Gamma et al., 1994] is created to execute the search; it visits every method invocation node (or other supported node type) and tests that node against the search criteria. By default, the search is conducted against the entire code base, as contained within the current project; Java files free of dangling references—as indicated by the absence of compilation errors—are fed directly into the AST visitor. Files containing dangling references are processed by the PPA tools to regenerate binding information and then sent to the AST visitor.

Scanning the entire code base one file at a time introduces a noticeable delay in Trident’s execution and was a usability issue commented on by all our participants. Standard refactorings circumvent this delay by using an optimized binding index that Eclipse generates during compilation. However, this index is unusable for dangling references and as such the performance optimization of Trident search remains an issue.

4.4.3 Restriction

The list of AST node search candidates collected by the AST visitor in the previous stage is presented to the developer in the Restriction stage. The node details are displayed in a list and a developer who wants additional details can double-click on any search candidate to bring up the associated Java file in the Eclipse editor with the line of interest highlighted.

This stage offers the first of many “escape routes” to the developer. He can choose to accept all the search candidates, further investigate candidates to select a few, or reject them all and return to the previous screen to reformulate his search. The ability to iteratively refine and selectively accept (or reject) search candidates helps offset some of the ambiguity inherent in the search process itself.

3 Earlier prototypes included the ability to limit the search to a few files but this feature was seldom used.
4.4.4 Refactoring Configuration

The refactoring dialog that appears at the Refactoring Configuration stage is customized to the kind of node being transformed. When a SimpleType or QualifiedType is being changed, a minimalist dialog reminiscent of the original Eclipse “Rename” refactoring is presented (see Figure 4.9).

When methods (or constructors) are being refactored, a dialog modelled after the “Change Method Signature” refactoring is presented. This stage of Trident is refactoring-like in that the user interface mimics the standard refactoring interfaces with which developers are already familiar. It is also refactoring-like in that it is built atop the Eclipse Refactoring Language Toolkit (LTK). Integrating Trident with the LTK allowed us to offer many of the other convenient features that users have come to expect of refactorings: when changing the name of a dangling method reference, Trident checks to ensure that the new name does not conflict with another method name in the same scope; manually entered data is also checked to ensure that it conforms to Java identifier and naming rules; it also assists with the provision of completion assistance features. The “Browse for Method” button (top right of Figure 4.5) and the analogous “Browse for Type” button (top right of Figure 4.9) allow the developer to quickly find replacement entities as needed. If, as noted earlier, a replacement method is selected using the “Browse for Method” button, the LTK helps to include the proper import statements as well. If the replacement method chosen is a static method, the expression name is updated to the name of the declaring class. Most importantly, using the LTK at this stage, as opposed to directly editing strings to affect changes, allows us to provide global undo functionality in the final stage in the event that the developer decides to roll back his changes. This is another “escape route” that we provide to the developer.

The Refactoring Configuration dialog of Figure 4.5 is unlike standard refactorings in the sense that either the method name or the method expression can be replaced with one or more
Figure 4.7: Selection assistance activated by attempting to select the argument list of the method.

private static final Category cat = Category.getInstance(ResourceBundleTest.class);

Figure 4.8: Selection assistance completed.

private static final Category cat = Category.getInstance(ResourceBundleTest.class);

Figure 4.9: SimpleType and QualifiedType refactoring configuration dialog.
Java expressions. Thus changes similar to the Replace Method Expression and Introduce Parameter Object outlined in section 3.4.2 can be enacted using Trident.

4.4.5 Preview and Enactment

The Preview and Enactment stage offers yet another “escape route” as developers can review the specified transformation and selectively remove some change locations from further consideration.

4.5 Summary

In this chapter, we presented an overview of the design and function of Trident, our prototype tool for refactoring references. Motivated by the work of Henkel and Diwan [2005] and Balaban et al. [2005] we envisioned a tool that: (1) allowed the developer to operate independently of the library designer and (2) offered functionality that was like standard refactorings (e.g., preview and undo) but was also (3) unlike standard refactorings in the types of changes it would permit (e.g., replace a method expression with an arbitrary expression). Some of the specific refactoring features we strived to emulate were that the tools should be (1) exemplar based, (2) offer ample opportunities to override or reverse changes proposed by the tool, and (3) provide completion assistance where possible. Finally we presented a walkthrough of Trident and mapped its 5-step process to our initial goals.

---

4This feature holds true for the refactoring configuration dialog of the other kinds of nodes as well.
Chapter 5

Case Studies

To determine whether even partial tool support for refactoring references is likely to be beneficial in practice, we conducted case studies with two industrial participants, in which they were asked to migrate a software system from dependence on an out-of-date library version to a more recent version. Each participant was asked to undertake the code migration first using the Trident tool, and then attempt the same migration again without the benefit of our tool support. Our research questions were: (RQ1) “How painful is it to refactor dangling references in source code using existing tool support?” and (RQ2) “Does Trident help developers reduce the difficulty of refactoring in these cases?”

Section 5.1 describes the methodology we used to conduct our case studies. Sections 5.2 and 5.3 describe our qualitative observations of the participants undertaking their refactoring tasks using Trident to assist them, and subsequently performing the same refactoring manually. Section 5.4 compares the approaches quantitatively and qualitatively, discussing the implications of the study.

5.1 Case Study Methodology

We recruited two industrial software developers to participate in our case study, and asked them to refactor references in a software system that was currently in an un compilable state due to changes within a depended-upon library. The goal of the refactorings was to restore the system to a compilable state.

Fellow graduate students were our pool of potential participants. From amongst this population we selected two participants based on (a) number of years of industrial (i.e. non-research)
software development experience and (b) some level of familiarity with Eclipse. The two participants ultimately chosen had the most industrial experience.

5.1.1 Systems

We chose to have the developers migrate the JaxMe project\(^1\), an open source implementation of the Java Architecture for XML Binding (JAXB) specification\(^2\), which defines how an XML schema can be transformed into a set of Java classes and interfaces for programmatic manipulation. The 1.63 release of JaxMe used in this study contains 213 classes and just under 18928 LOC (according to “\texttt{wc -l}”). JaxMe was a strong candidate for our study as it relied heavily on a library which had evolved significantly, with functionality deprecated and removed: the Apache log4j library.

Log4j\(^3\) provides application logging functionality to developers. In place of \texttt{System.out} or \texttt{System.err} calls to write logging and error messages to the console, log4j provides a runtime configurable logging framework which can selectively disable specified levels of logging during execution, and can provide output in multiple formats. As log4j has increased in both popularity and maturity, its API has undergone significant changes. For example, in previous versions of the library, developers would use the \texttt{Category} class to access the “category” of functionality that they wished to log. However, in the transition from versions 1.2.8 to 1.3, the functionality provided by the \texttt{Category} class was largely supplanted by the \texttt{Logger} class, because using a “logger” for logging made more sense than using a “category”.\(^4\) Similarly, the \texttt{Priority} class was originally used by developers to define the importance of a logging message being sent to the \texttt{Category} objects used for logging. In the new version, the type was replaced with a new \texttt{Level} class which allowed developers to set the \texttt{error level} of logging messages sent.

\(^{1}\)http://ws.apache.org/jaxme [last accessed 28 April 2010]; version 1.63 was used.
JaxMe makes extensive use of the log4j library, which consequently meant that these conceptually simple alterations to the log4j API had widespread consequences in the source code.

5.1.2 Task

For the case study, each developer was asked to migrate the JaxMe project between versions 1.2.8 and 1.3 of the log4j library. Only a subset of the real transition was allowed to cause compilation problems: the change of Category instances to Logger; and the change of Priority instances to Level. The full specification of the needed transformations is shown in Table 5.1.

There are several other differences between the 1.2.8 and 1.3 versions of log4j, but we elected to restrict our study to the above subset for three reasons: (1) the time required of participants to perform the case study had to be restricted for practicality; (2) these classes, and their associated methods and/or fields, constitute the majority of the differences between the two versions; and (3) the classes that have been removed and their intended replacements are clearly indicated in the library’s documentation, through the use of the @deprecated and @see tags, thereby eliminating any ambiguity about what modifications are needed.

We simulated the partial transition to 1.3 by providing stubs for missing entities that were not in the limited subset.

5.1.3 Setup

Participants were provided with the Eclipse IDE\(^5\), configured with the Trident tool as a plugin. At the start of the case study, each participant was provided with a short tutorial introducing them to the purpose and usage of the Trident tool. The tutorial showed a dozen unique sample scenarios in which the Trident tool was used to find and refactor code containing dangling references, to illustrate to the participant how the tool can be used.

\(^5\)http://www.eclipse.org [last accessed 28 April 2010]; version 3.5 was used.
Table 5.1: The full set of changes needed to port JaxMe from log4j version 1.2.8 to version 1.3.

At the completion of their training, the JaxMe v1.63 source code was loaded in their project workspace, with log4j v1.3 (adjusted as described above) in the JaxMe project’s classpath. After the initial attempt to build the JaxMe project, Eclipse would inform the developers that exactly 100 compilation errors resulted—due solely to the absence of the old log4j library. Participants were then asked to modify the source code to eliminate the dangling references left by the log4j library transition, and reduce the compilation error count to 0. To alleviate the developer’s need to familiarize themselves with the log4j documentation, participants were provided with a concise list of the source entities that had been removed between versions, and the names of the entities that should be used to replace them replacements, extracted and condensed from the log4j documentation.

Participants undertook this task twice. On their first attempt (the tool treatment), they were to use the Trident tool, and on their second attempt (the manual treatment) they were to attempt the change “manually” using any functionality within Eclipse they desired, but without the assistance of Trident. In both treatments, we asked the study participants to “think aloud” as each went through the task to help us understand their thinking process, action rationales, and
reactions to the results. While each treatment was conducted on the same system and problem, the treatments were conducted 24–72 hours apart: our goal in doing this was to partially reduce the learning effect caused by having to repeat the same tasks, since we expected that this would prevent developers from remembering every detail of the task, thus potentially obscuring our attempt to measure differences between Trident and existing techniques. We did expect though that this would not eliminate the learning affect, and thus give a small advantage to developer performance in the manual treatment.

For the tool treatments, participants were asked to use only Trident for any automated/semi-automated refactorings to the code: they were allowed to make manual modifications to the source code as they desired (i.e., via the source code editor). They were also allowed to ask the study investigator about any details of Trident’s usage, but not details of how the code could or should be modified to successfully complete the task.

For the manual treatments, each participant was allowed to use only those tools provided within the Eclipse IDE (e.g., refactorings, lexical search/replace tools) and manual modification to alter the JaxMe source. Participants were again free to ask for assistance in using any of the tools provided by the Eclipse IDE.

5.2 Participant 1

The participant for the first case study described himself as having 9 years of experience in developing Java software, and had also used the Eclipse IDE for the past five years.

5.2.1 Tool Treatment

The participant began the tool treatment by reviewing the list of API changes provided with the intention of grouping together changes of the same type or same degree of difficulty so they could be attempted together. He decided to begin by changing all the dangling field references
of the Priority class into fields of the same name in the new Level class. Based on the high degree of lexical similarity between the names, he reasoned that this would be a simple case of string replacement and thus most likely to succeed.

Having decided on a group of API changes to tackle together, he began to iteratively apply the tool to enact the change. He selected a Priority.ERROR field reference as his first exemplar, and used Trident to perform a type search on the JaxMe code. Since these dangling fields followed the Java programming-style convention of using all upper case characters for describing constant variables and enums, and he had not yet examined the JavaDoc associated with the Priority class, he assumed the dangling field references referred to a Java enumeration type. As this assumption was incorrect, the Trident type search yielded no results. His next attempt was to perform a type search on the ERROR field name. This search retrieved all simple names of type Priority which included the desired result, but also local variable declaration statements (e.g., org.apache.log4j.Priority p;). He tentatively accepted the excessively broad search results and moved on to the refactoring page. Here the participant realized that since his search was launched using the ERROR simple name he would only be able to change this suffix (i.e., the field name), when in fact he wanted to change the qualifying prefix (i.e., the type containing the field, specifically from Priority to Level). After these false starts, the participant performed a verbatim search on the fully qualified name of the type and field (org.apache.log4j.Priority.ERROR) which yielded the correct results. Since a qualified name was his exemplar for Trident, he was able to alter the entire name to point to a new field reference with the aid of a dialog box in the refactoring screen. The same strategy was employed to correct all the other dangling references pointing to fields in the Priority class.

With the corrections to field names complete, the participant began tackling the dangling method references. Referring back to the list of API changes, he settled on Category.get-

Root() to Logger.getRootLogger() as the next change to address. He began by selecting the method name `getRoot()` to initiate a search with Trident, and then decided that other unrelated `getRoot()` methods might be used in different contexts in the system. Accordingly he revised his exemplar selection to `cat.getRoot()`, in which `cat` is an instance variable of type `Category`. (The `getRoot()` method is static, and while the practice is discouraged, JaxMe sometimes accesses such static methods through an object of that type, as is the case here.) He used Trident to perform a verbatim search on the method name and a type search on the method expression. Satisfied with the search results, he proceeded with the refactoring, and in the final preview screen he was pleased to note that though his exemplar had been `cat.getRoot()` (with the intention of transforming it to `cat.getRootLogger()`), by employing a type search on the method expression, he had also captured direct class invocations of the form `Category.getRoot()`, and these were also being correctly refactored to `Logger.getRootLogger()`.

For the next API change, he chose to transform `Category.setPriority(...)` to `Level.setLevel(...)`. Following the previous example, he conducted a type search on the method expression, and a verbatim search on the method name. In this case, the participant noticed that unlike the previous cases he dealt with, `setPriority(...)` takes an argument. He decided to not have Trident restrict the search based on parameter types and numbers; he said that he wanted to ignore the arguments because he wanted “to use Trident to find the different kinds of arguments being passed in, before I decide on what changes to make.” He suspected that a fully qualified field name is the most common argument supplied to `setPriority()` based on what he had observed in the JaxMe source code, but was not sure that this covered all of the cases in which the method was used. The participant chose `Category.setPriority(org.apache.log4j.Level.DEBUG)` as his exemplar input into Trident, and saw that other cases exist where the argument is a local variable reference, or the result of a method invocation. He decided to accept that the method arguments
varied, and chose to have Trident replace all `Category.setPriority()` invocations with `Level.setLevel()`, without altering in any way the arguments supplied to the original method invocation. He reasoned that this ought to deal with the majority of the situations, allowing him to then address the hopefully smaller number of compilation errors that should arise due to inconsistent parameters. After Trident performed the refactoring, the participant found he only had a handful of compilation errors left to deal with. After looking at the error list, he decided that, while he could use Trident to fix the remaining cases, each case differed sufficiently that he would not be able to use to Trident to refactor more than one dangling reference at a time, eliminating the effort savings of using the tool; he instead chose to manually locate and edit the final locations.

5.2.2 Manual Treatment

The participant returned three days later to perform the manual treatment of the case study. He was provided with the Eclipse IDE again, loaded with the JaxMe software system in the same state as it was prior to the start of the Tool Treatment, but with the Trident plugin removed. As in the tool treatment, he began by attempting to transform dangling references to fields on `Priority` into field references on `Level`. Based on the lexical similarity between each dangling reference and its replacement, he decided to use Java Search in Eclipse to find and replace the dangling `Priority` field references with their equivalent `Level` reference. He decided to start by replacing all strings of the form "org.apache.log4j.Priority. DEBUG", but discovered that Java search performs only search, there is no provision for replacing the search matches with a new string. Undeterred, the participant resorted to using Eclipse’s File Search and Replace functionality, which he noted would allow him to write regular expression patterns to do the matching with more flexibility than a standard find-and-replace search.

The first four attempts at creating a regular expression for locating `Priority` references
failed to return any results, despite the fact that Eclipse File Search provides inline regular expression assistance. Two errors arose while trying to determine if the periods in the qualified name should be escaped\(^7\), and two more errors arose because of whitespace mismatches. He finally created a pattern ("org.apache.log4j.Priority.(\w+)") to match the common type qualifier for the field names, while accounting for the differences in the actual field names by simply capturing the field name with \w+. He wrapped the last part of the pattern in parentheses to designate that portion of the pattern as a capturing group, to use in the replacement string. He then crafted a replacement expression ("org.apache.log4j.Level.\1") that used the capture group defined above to append the field name at the end of the Priority matches (e.g., DEBUG from org.apache.log4j.Priority.DEBUG) to a new string while replacing “Priority” with “Level”. He then had Eclipse show him a preview of the proposed alterations, and decided to enact the search and replace. Afterwards, he discovered unexpected effects:

- In JaxMe’s Main class, the local variable declaration org.apache.log4j.Priority p was transformed into a field access statement: org.apache.log4j.Level.p. The pattern he wrote captured the space separating the type from the instantiation in the declaration, resulting in the local variable’s name being appended as a field access. Because regular expressions have no syntactic or semantic awareness, the search was not aware that the lack of a period after Priority, and its replacement instead with a space, was significant.

- Static method calls to org.apache.log4j.Priority.setPriority(...) were replaced with org.apache.log4j.Level.setPriority(...). He noted that although this change could work as an intermediate step in the eventual trans-

\(^7\)The participant was initially confused since, in this case, escaping the dot (\.') operator had no discernible effect on the pattern matching. The dot operator matches any single character, while an escaped dot operator (\\.') matches only a period, yet both seemed at first to be equally effective in the JaxMe system when searching against fully qualified Java type names.
formation of `org.apache.log4j.Priority.setPriority(...)` to `org.apache.log4j.Level.setLevel(...)`, it was an unintended side effect of the replacement and not his intention.

These consequences prompted him to comment, “The big problem with `grep` and regular expressions is that you capture things that you don’t expect”.

He remarked that the compilation error count in the project barely decreased at this point, with many of the files displaying errors at the locations where the string replacement has taken place. After investigating, he discovered that the problem was that the `Level` type could not be resolved, since the appropriate import statement was missing from the affected classes. To fix this, the developer used Eclipse’s Organize Imports feature to automatically resolve unknown type declarations by importing the appropriate classes.

Next, the participant attempted to replace calls to `Category.getInstance(...)` with `Logger.getLogger(...)` using File Search and Replace. Using a similar regular expression pattern as before, he was successful in enacting the change, but again encountered the problem that `Logger` was an unresolved type. This particular string replacement was widespread, affecting 44 files and causing a dramatic boost to the compilation error count. Employing the same strategy as before, he invoked Eclipse’s Organize Imports feature, but this time received an error message for each file: “ambiguous references, user interaction is required.” The problem was that two different yet semantically similar `Logger` classes existed in the project’s classpath: the `org.apache.log4j.Logger` class which the participant was trying to use, and the `java.util.Logger` class which is part of the standard Java Software Development Kit. Unable to distinguish between which of the two identically named classes should be imported, Eclipse is unable to automatically handle this for the participant.

The participant was not keen on manually inserting import statements in every file, and decided to look at a few of the errors to see if there was a workaround he could attempt. He noticed that in the JaxMe code base, invocations of `Category.getInstance(...)`, `Logger.getLogger(...)`, and `Category.getInstance(...)` were made.
return a singleton object usually assigned to a static member variable. The format of this statement was, after his previous transformations, `public static final Category cat = Logger.getLogger(...);` the participant wondered if maybe the assignment of `Logger` to type `Category` was confusing Eclipse’s Organize Imports feature. He wrote a regular expression to change such declarations to `public static final Logger cat = Logger.getLogger(...);`, but this boosted the compilation error count to 240, as it created more locations in the code where the `Logger` type could not be resolved. He undid the change, and decided that import statements had to be added first.

To avoid manually editing every file, the participant decided to write a regular expression that would recognize the package declaration at the start of a Java file, adding an import statement for the `Logger` class afterwards. To do this, he created the search expression `package\s+\S+` to find all package declarations in files, and the replacement expression `\0 import org.apache.log4j.Logger;` to append the import statements to the end of the package declaration. In doing so, the participant acknowledged that the import statement would be added even to classes which did not need it, but he reasoned it would be easy to clean up afterwards using the Eclipse tooling. Enacting this change dropped the error count from 240 to 42, fixing the unresolved imports issue. He then ran Organize Imports to eliminate those import statements that had been unnecessarily introduced.

However, he also introduced a new problem: in the `JavaSource` class, a string literal happened to match the participant’s regular expression. The participant noticed a new compilation error in the list which was a syntax error, making it stand out from the other problems. On investigation, he recognized that the code snippet `result.append("package ");` in the `JavaSource` class was detected as a match by his original regular expression pattern. As a result, the line had been altered to read `result.append("package "); import org.apache.log4j.Logger;`, creating a syntax error that he manually fixed.

He then transformed calls to the static method `Category.getRoot()` into `Logger.`
getRootLogger(), again noticing JaxMe’s inconsistent treatment of static method invocations: sometimes it invokes static methods via a call on an instance variable (e.g., `cat.getRoot()`), while in others the method is invoked directly from the class (e.g., `Category.getRoot()`). Though the two cases are semantically similar, each required its own regular expression pattern to resolve. The participant noted that his replacement for `getRoot()` on static `Category` variables relied heavily on the code convention that all such variables were named `cat`. If `Category` variables had been declared under a number of different names (e.g., “category”, “myCategory”, etc.) this change would have been much more difficult.

On reaching this point, the developer had only 8 errors left, each of which were sufficiently different that he decided to manually fix each problem.

5.3 Participant 2

The participant for the second case study described himself as having 7 years of experience developing Java software, and had used the Eclipse IDE for two years.

5.3.1 Tool Treatment

At first, the participant had difficulty in selecting the proper exemplar with which to initiate the Trident tool. In attempting to transform `org.apache.log4j.Priority.DEBUG` into `org.apache.log4j.Level.DEBUG`, he selected just the simple name `DEBUG` and performed a verbatim search for other simple names with the same string pattern. When the search returned only two matches, he was suspicious and decided to broaden his scope to a verbatim search for all qualified names containing `org.apache.log4j.Priority`. After seeing the large number of search results returned, he requested assistance from the investigator. After an impromptu tutorial on how exemplars should be chosen, he was able to quickly refactor all the fully qualified dangling field references to the `Level` class.
He then noted in the error view that an unqualified field reference to \texttt{Priority.WARN} had not been refactored. Rather than use Trident, he decided to manually correct this single case by overwriting \texttt{Priority} with \texttt{Level}, and using Eclipse’s Quick Assist to resolve and import the class. This resulted in another compile error stating that \texttt{WARN} could not be resolved. The participant’s confusion was resolved when he realized that there were multiple classes with the name \texttt{Level} on the class path, and he had imported the wrong one.

The participant continued by refactoring calls on \texttt{Category.getInstance(...)} to instead call \texttt{Logger.getLogger(...)}, and noted that \texttt{getInstance()} is static and is present in three different overloaded formats. Based on this information, he decided to do a type search on the method expression, a verbatim search on the method name and ignore all the arguments. Trident returned 44 matches, and upon enacting the change from \texttt{Category} to \texttt{Logger}, the error count dropped to 37.

Next, he considered changing \texttt{Category.getRoot()} to \texttt{Logger.getRootLogger()} by applying the same search pattern he had used for \texttt{Category.getInstance()}. On examining the search results, he noted that although he intended to look only at invocations of \texttt{getRoot()} occurring directly on the \texttt{Category} class, some static instance variables (of type \texttt{Category}) were also captured in the search. Consequently, the participant was uncertain about the correctness of his anticipated change, and wondered if he should examine each search result to filter out all the cases he did not expect to find, and handle such cases individually. He decided instead to refactor all the search results since Trident provides global “undo” functionality, and can revert the refactoring if it causes problems. On seeing, in the final preview pane, that Trident introduced comments above all the new \texttt{<localvariable>\_getRootLogger()} statements to make it clear that a static method was being invoked on an instance variable, he was encouraged and thought that it was highly likely this was the refactoring he wished to do.

Finally, the participant converted all the static variable declarations of the form \texttt{public
static final Category cat = ... into variable declarations of the form public
static final Logger cat = ..., by selecting a Category reference as his exemplar and performing a type search in Trident to find other locations where Category is part of a simple type reference. He asked the investigator if it was possible to restrict the search only to type declarations within variable declaration statements (as opposed to catch clauses, method declarations, etc), but was informed that context restricted searches were not yet supported. He continued with the refactoring, reducing the error count to 5. At this point, he decided to fix the remainder manually.

5.3.2 Manual Treatment

The participant was brought back the following day to perform the manual treatment of the case study. He was provided with the Eclipse IDE again, loaded with the JaxMe software system in the same state as it was prior to the start of the Tool Treatment, with the Trident plugin removed.

He began by reasoning that the API change of org.apache.log4j.Priority to org.apache.log4j.Level represented a refactoring that moved fields from one type to another. Upon examining the list of Eclipse refactorings for a suitable match, he activated the “Move” to attempt to enact this refactoring automatically, and was greeted with an error message that stated, “Destination type does not exist.” He realized that Move refactorings are intended to move declarations and not references, and that none of the Eclipse refactorings can be applied to dangling references.

With refactorings clearly unavailable, he decided to try Eclipse’s Java Search tooling, but was confused by the lack of a replace option. After spending a few minutes examining every Eclipse search menu he could find, he eventually located a lexical replacement option under Eclipse’s File Search tool. He then proceeded to use case sensitive search to find and correctly replace all of the Priority dangling field references with the Level type.
In an attempt to quickly reduce the task to a more manageable size, he noted that the API change `Category.getInstance(...)` to `Logger.getLogger(...)` was the most common source of errors, and he decided to attempt it next. Using case sensitive search on `Category.getInstance()`, he replaced it with `Logger.getLogger()`. Expressing trepidation at the size of the change, the participant conducted an exhaustive review of every search match and change in the text preview screen. After satisfying himself with the accuracy of the change, he enacted the modification, and was surprised to find that almost all the errors remained. He recognized that the errors were now because the new `Logger` type could not be resolved, and so tried to apply Eclipse’s “Quick Fix” to the entire group of errors in the problems view, but could not as it can only be applied to individual errors. He decided to resort to opening each of the 37 affected files one by one, and adding an import statement to address the problem. The tedium of this task prompted him to say that “I am beginning to realize what a pain this really is. Initially I had thought Eclipse search and replace would do this for me.” As he came close to finishing, he remembered that there was a refactoring scripts option in Eclipse, which he hoped could be adapted to create a macro to complete this task for him. However, he quickly realized that for refactoring scripts to work, the underlying Eclipse refactorings must also work; a precondition that does not hold with dangling references. Resignedly, he finished the manual modification.

The next problem the participant addressed was the change of `Category.getRoot()` to `Logger.getRootLogger()`. As previously noted, the `getRoot()` method is sometimes invoked on static instance variables, and sometimes invoked directly on the class. Fortunately, in all of the cases involving static instance variables, the name of the instance variable is consistently “cat”, thus allowing for a relatively easy lexical search and replace. He used the same string search and replace to convert `Category.getRoot()` style method invocations into `Logger.getRootLogger()`, but made a copy-and-paste error in the replacement field, causing an extra, empty pair of parentheses to be attached to the end of the new method in-
vocation. He attempted to undo this change, but found that the Eclipse File Search tooling does not provide global undo capability; the participant needed to undertake a second round of search-and-replace to fix his previous mistake.

The last major change that remained for him was transforming the instance variables of the form `static Category cat = ...` into `static Logger cat = ....` However, his previous refactorings had partially altered the right hand side of these statements so they were now found as `public static Category cat = Logger.getLogger(...)`. The participant decided that a regular expression was his best strategy to enact the change in this case. Using the left hand side of the assignment as an “anchor”, he wrote the regular expressions “`Category\s+cat\s*=\s*Logger.getLogger`” to search for matches, and “`Logger\s+cat\s*=\s*Logger.getLogger()`” as the pattern to replace it with. With the change complete, he saw the error count jump from 34 to 66, and realized he had intended to use the replacement expression `Logger\s+cat\s*=\s*Logger.getLogger()`. Since undo functionality is not available, he decided to try another round of search-and-replace using “`Logger.getLogger()`” as his search expression, but committed yet another copy-and-paste error using “`Logger.getLogger()`” as his replacement. This error might have been caught in the text preview pane, but he accidently hit the “Update” button instead of “Preview”, thus committing the change to the workspace, and causing the error count to jump to 72 as both the method name and the method expression are incorrect. At this point, he exclaimed, “This is a nightmare. Is it okay if I quit this task?”

The participant decided to continue working, but ran into more difficulties as he tried to continue to fix his previous errors by using `Logger.getLogger` as the search expression, and replacing it with `Logger.getLogger()`. However, the absence of the parenthesis in the first pattern, and its presence in the second, created cases like `Logger.getLogger(()` which further increased the error count to 83. Gradually and carefully, the participant managed to fix the consequences of this series of errors, and returned to an error count of 23. He decided
at this point to manually correct all the remaining compiler errors, staying clear of regular expression-based tooling. Finally, he reduced the error count to 0, noting that “the task seemed easy but the [manual change process] was really messy and I am not confident in the solution”.

5.4 Results and Analysis

For the tool treatments, the participants began Trident invocations 28 and 21 times, respectively, and took these to completion 14 and 10 times, respectively; they manually modified files 7 and 5 times, respectively, in addition. Participant 1 required 57 minutes to complete the treatment, with Participant 2 requiring 100 minutes (note that these timings include waiting for the tooling to complete, and interaction with the observer).

For the manual treatments, the participants manually modified files 40 and 104 times, respectively. Participant 1 required 74 minutes to complete the treatment, 130 minutes for Participant 2.

We analyze our findings in terms of our research questions, below.

5.4.1 Research Question #1

How painful is it to refactor dangling references in source code using existing tool support? Both participants struggled at times to refactor source code with dangling references, but their difficulties varied dramatically between them. Participant 1 had a solid grasp of regular expressions, using them to great advantage, perhaps providing the best possible example of state-of-the-practice tool support. He seemed to require very few operations to complete the task, and in the end elected to manually fix a few compilation errors directly. However, he still was not able to write patterns that were completely error free; in most operations, participant 1 had one or more unanticipated side effects occur that needed to be addressed manually. It was also necessary for him to do what he described as “an ugly hack” in which import statements
were inserted globally across the system to address a class resolution problem affecting only a portion of the system. This participant’s breadth of experience with Eclipse (as both a user and plugin developer), combined with his depth of knowledge about regular expressions made him an excellent candidate to test our approach. His difficulty in completing the assigned task despite these advantages speaks to the limitations of Eclipse and regular expression-based support in addressing this problem.

By contrast, Participant 2 had significant difficulty in the refactoring task, to the point where he contemplated abandoning the migration of the JaxMe system between log4j versions. His difficulties reflect many of the problems inherent with such tooling. He was not able to use the Eclipse refactoring tools with which he was comfortable, because they were not designed with refactoring dangling references in mind. He wanted to use Eclipse’s Java Search functionality to find and replace Java types and fields, but the search inexplicably has no replace option. He had difficulty finding the file-based search-and-replace tool he wanted, and when using it, ran into numerous problems caused by typos or copy-and-paste bugs that were further complicated by being unable to undo his mistakes and start over. In the end, Participant 2 spent a considerable portion of time manually enacting changes to the code.

Participant 2 also noted that his experience could have been worse; JaxMe appeared to consistently use a naming convention when declaring Category variables in the code, which allowed him to leverage that pattern (specifically, Category cat) to make the task of finding correct pattern matches easier. Had this convention not been in use (e.g., each variable declaration used a different name), he felt his task would have grown in difficulty.

Both participants had common problems which we feel are worth noting: both expressed frustration when refactorings attempted with standard Eclipse tooling had no effect on reducing the compilation error count, or caused increases in the error count. In many cases, refactorings required multiple steps before an error reduction would occur, causing them to wonder if their actions were actually having any affect.
5.4.2 Research Question #2

Does Trident help developers reduce the difficulty of refactoring in these cases? Both participants strongly stated that the task was far easier with the assistance of the Trident tool.

Participant 1 found that Trident’s preview window, combined with its undo functionality, allowed him to attempt refactorings in an exploratory manner. He would often use the preview window to look for specific cases in which he wanted to ensure that his search criteria worked, and gain early feedback as to what problems could exist. In cases where he was not sure if he had captured all the dangling references of interest, or had captured too much, he would often proceed with the refactoring, since he was confident that Trident’s undo functionality would allow him to revert easily back if he was wrong.

Participant 2 by comparison was far more deliberate about enacting changes, and as such heavily relied on Trident’s preview window to understand how his searches were working, and to ensure that his expectations as to the tool’s refactoring would match reality. In cases where he was confused or unsure about how a particular refactoring might affect code, the comments shown in the code previews provided sufficient feedback to encourage him that he was on the right track.

Both participants seemed to make steady progress with the Trident tool. Neither participant saw an increase in the number of compilation errors after performing a particular refactoring, and both made steady, consistent progress in reducing the number of compilation errors in the JaxMe system with every Trident invocation.

5.5 Summary

To evaluate the efficacy of Trident in supporting the refactoring of references during API migration, we conducted a case study involving two industrial participants. We sought to answer two research questions: (RQ1) “How painful is it to refactor dangling references in source code
using existing tool support?” and (RQ2) “Does Trident help developers reduce the difficulty of refactoring in these cases?” To this end participants were asked to migrate the open-source JaxMe system from using version 1.2.8 to 1.3 of the log4j logging library. Each developer attempted the task with Trident support (tool treatment) and then repeated the task using standard tooling (manual treatment). From our detailed qualitative and quantitative observations a number of results emerged.

When attempting to refactor references, both participants experienced difficulty. Participant #1 was proficient with regular expressions allowing him to fully utilize Eclipse tooling. However, he struggled to craft a regular expression for each context that properly accounted for whitespace and special characters, often captured additional strings unintentionally, and, on one occasion, had to resort to what he called an “ugly hack” to correct import statements. Participant #2 experienced even greater difficulty because he was less proficient with regular expressions which meant he resorted to making many changes by hand; at one point repeating the same editing operation on 37 files. He also experienced numerous copy-and-paste errors that were exacerbated by the inability to undo his mistakes and start over. At one point, his frustration was so great he asked; “Is it okay if I quit this task?”. Participants 1 and 2 took 74 and 130 minutes, respectively to complete the API migration manually.

When carrying out the same task with Trident tool support, both participants stated the task was much easier. Participant #1 like the ability to investigate and enact changes in a stepwise manner, knowing that he could override the tool or undo his changes at any time. As he described it, “instead of jumping of a cliff there were plenty of points to dangle a rope off the cliff and see if that [change] is what you wanted”. Participant #2 experienced fewer difficulties using Trident compared to his manual attempt. Though he was more cautious than participant #1 when using the tool, he made steady progress at decreasing the number of compilation errors which at the end prompted him to say, “If dealing with API changes [Trident] is definitely useful”. Participants 1 and 2 took 57 and 100 minutes respectively to complete the API migration.
task with Trident tool support.

A final point to note is that the times reported for both studies include time spent engaged with the researcher. Moreover, the task time reported for Trident includes tool execution time which amounted to several minutes over the course of the study. With performance enhancements to Trident we would expect this time to decrease.
Chapter 6

Discussion

In this chapter, we examine a number of additional points, outstanding issues, and avenues for future work.

6.1 Threats to Validity

Having each participant repeat the same migration task may call into question the validity of the results from the second treatment, since learning effects could accrue. However, we wished to put our approach up against the toughest comparison: industrial-strength tooling when the participant was already familiar with the specific task. Both qualitatively and quantitatively, Trident outperformed the standard tooling, despite biasing the study strongly in favour of the standard tooling.

It is possible that participants’ exposure to the Trident tool shaped their approach in the manual treatment such that the nature of the refactorings they attempted were not appropriate for their context; however, our observations show that the participants clearly understood what they were trying to achieve, and the problems they encountered stemmed largely from the inadequacies inherent in existing tool support.

Our case study demonstrates selection bias in two ways: first, in the nature of the systems examined, and second in the skill sets of the participants. The evolution of the log4j system between versions appears to be, in many respects, trivial. In most cases, a single class in the old version of the API needs to be replaced by a functionally equivalent class in the new version, and all this entails is a type change in the code: existing method invocations and field access on
these types are syntactically identical across both versions. In fact, documentation\(^1\) describing how code using version 1.2 of the log4j library should transition to version 1.3 indicates that:

For 99.99\% of users, [this transition] translates to the following string find-and-replace operations:

1. Replace the string “Category.getInstance” with the string “Logger.getLogger”.
2. Replace the string “Category.getRoot” with the string “Logger.getRootLogger”.
3. Replace the string “Category” with the string “Logger”.
4. Replace the string “Priority” with the string “Level”.

However, our case study participants demonstrated in their manual refactorings that this advice is simplistic. The JaxMe system does not always access static methods through their associated class, but may instead access them through an object instantiation of that class, which would not be caught by ‘search and replace’. Further, we note that a string replace on words such as Category and Priority indiscriminately across a system can also have serious side effects should those words be used in source code documentation, variable, parameter, or method names, or even in a string literal as Participant 1 discovered. Consequently, we argue that, while the evolution of log4j in this case may seem trivial, its usage within the JaxMe system makes migration between library versions a non-trivial matter for developers to resolve. In larger systems, using more complicated libraries than those simply providing logging functionality, we would expect this problem to be even worse, and the need for effective tool support greater.

We do note that there seemed to be a disparity in how each participant used the tools available in the Eclipse IDE, stemming from their differing skill-levels with respect to regular

\(^1\)http://articles.qos.ch/preparingFor13.html [last accessed 28 April 2010].
expressions. Participant 2 benefited the most from the Trident tool support as opposed to manual approaches in accomplishing their task. This could suggest that our choice of participants may unfairly paint Trident in a more flattering light. However, despite Participant 1’s skill at the manual treatment, he still ran into a number of serious issues which required “hacks” to address. Regardless of the skill-level of a developer, many aspects of refactoring references are not addressable by current tool support.

In considering how our results generalize, we are careful to note that we have conducted only two case studies with two participants, and both of these case studies were undertaken on the same system and with the same, structural problem and context. The nature of how APIs evolve may vary wildly, and log4j should not be considered as being archetypical of such evolution. Similarly, the manner in which JaxMe is affected by changes in one of its libraries’ evolution is likely different from many other software systems; JaxMe, in particular, has certain common patterns in how the log4j library was used which made refactoring dangling references easier in some cases, and harder in others, than it might have otherwise been. Many of the specific observations we made during the case study would likely change had any of these specifics changed.

However, while the kinds and nature of dangling references caused by library evolution may vary dramatically across specific systems, our case study—and our study of API change—demonstrates that they do occur, and that existing tool support does little to help developers to address the problems inherent in trying to refactor references. Trident has shown that it can help in some of these cases, and has the potential to be improved upon to handle cases that it currently cannot. As we explore how libraries evolve, and the kinds of tool support needed to support library migration in these cases, Trident’s potential usefulness should grow.
6.2 Tool Limitations

API changes may exhibit a 1:1 correspondence between replacee and replacer source entities (e.g., $A \times ()$ becomes $A \cdot y()$) or API changes can show $N:1$, $1:N$ (e.g., a facade class replaces many individual classes or vice versa) or even $N:N$ correspondence. From the outset, we have restricted the development of Trident to address just 1:1 API changes. We limit ourselves to this scenario because Trident is intended as a prototype tool to investigate the potential of our approach. Accounting for more complex refactoring scenarios is an area for future work.

6.3 Usability

Both participants had difficulty at times in selecting an exemplar from the source code that was appropriate for configuring Trident. For example, participants would often select the single word forming an identifier, which suggested to Trident that the participant was intending to invoke a search on a simple name, when the participant was in fact interested in operating on the type associated with that name, but had not selected enough of the identifier’s context. A straightforward solution to this usability issue is to prompt participants when the simple name selected is within a larger context that might be of greater interest. It should be noted that providing users with assistance in selecting the right code with which to initiate a refactoring is also an issue within Eclipse [Murphy-Hill and Black, 2008].

By default Trident scans the entire code base during the search query stage, which introduces a perceptible time delay in the use of the tool; Trident is not optimized in the least, at present. Early attempts at optimization have shown that it may be possible to significantly reduce this delay by creating a separate index for dangling references alongside the normal binding index that Eclipse already generates to boost refactoring speed. Another option is to provide developers with the option of limiting the number of files—by name or by package— included in the search; this could help alleviate the problem by reducing the quantity of code
searched. We deferred these enhancements because we felt investigating the usefulness of the approach took priority over improving its performance.

6.4 Future Work

Possible extensions to the research include enhancements to Trident and general improvements to the process by which developers adapt to API change. With regard to improving Trident the second participant made the interesting suggestion that Trident should be integrated with the Eclipse “Quick Fix” feature: when a method cannot be resolved, Eclipse already offers an option to create the method and Participant 2 advocated for a new option such that “if a method was missing you could repoint it somewhere else”. He noted that integrating Trident with Quick Fix would enhance tool usability since the developer would be “presented with a solution right alongside the problem”. Another way of incorporating the ideas behind Trident directly into Eclipse would be by offering an option in each refactoring to modify references but leave the declaration alone. For instance, at present, the rename refactoring assumes all method references must be updated when a method is renamed. Eliminating this assumption and allowing the developer to specify which references should be updated would be a good step forward.

At present, Trident support is principally restricted to the modification of dangling methods and renaming of entities. There remain numerous breaking changes that Trident does not assist with but which do occur with some regularity in practice; the most common unsupported changes are: a different return type, different exceptions thrown, and returning a collection of objects in place of a single object (i.e., $\text{Object[]} \, \text{vs. Object}$). We avoided such use cases because of the complexity of the changes involved, but it is because of this complexity that tool support would be a boon. Nita and Notkin [2010] reiterate this point: “one of the biggest hurdles we encountered [in our research] ... was suitable handling of try/catch blocks”.
Trident tackles the problem of missing binding information by recreating that information with the help of PPA [Dagenais and Hendren, 2008]. This particular solution works for API migration scenarios because the code base is largely intact and thus the PPA inference algorithms are effective. However, in other dangling reference scenarios, such as test-driven development, the code base is mostly incomplete and PPA would be far less effective. For instance, developers following test-driven development are—in our experience—unlikely to write complete import statements for the classes being designed and, as such, the PPA strategy of using import (mentioned in Section 4.4) would be rendered ineffectual. Employing Trident in the other dangling reference usage scenarios mentioned in Section 1 is likely to require a substantial modification of the implementation details.

Another valuable enhancement to Trident would be allowing for parameterized replacements as first described in Section 4.1. Thus transformations of the form:

\[
\text{myVector.indexOf(obj, 1)} \rightarrow \text{Util.indexOf(myVector, obj, 1)}
\]

could be replaced with transformations such as:

\[
\text{<?>indexOf(obj, 1)} \rightarrow \text{Util.indexOf(<?>}, \text{obj, 1)}
\]

In the latter case if the instance variable were named something other than myVector (e.g., aVector, vector1, etc.), the transformation would still be successful.

Enlarging our case study to include more class libraries and more target systems would provide greater support for the external validity of this work, as would an increase in the number of participants. As a next step, we intend to perform a formal experiment into the effectiveness of the approach.

One possible general improvement to managing library changes is improving the DRR cycle. The Java language specification [Gosling et al., 2005] has reserved some keywords (e.g., const, goto) in advance of any implementation support for such term because (a) they anticipate these terms may be used and (b) they recognized that subsequently altering the list
of reserved words could cause problems for all developers. Similarly, the idea that API designers could defensively declare their intentions in advance, forms the basis for the work on *anti-deprecation* by Spoon [2006]. Deprecation allows library designers to signal removal of a method (or other entity) but “sometimes a future version of an interface will require an additional method. An annotation for such future required methods could be called *anti-deprecation*”.

More sophisticated improvements to managing API evolution could come from combining multiple research approaches together. A list of breaking changes from Eclipse API Tooling could be passed to API recommendation tools like Diff-CatchUp [Xing and Stroulia, 2007] and SemDiff [Dagenais and Robillard, 2008] and be used to identify replacements. These change details could then be written down in a commonly agreed upon formal specification [Chow and Notkin, 1994, Balaban et al., 2005, Nita and Notkin, 2010, Tip et al., 2003] that would be included in each new library version; similar to the manifest.mf file that is already included in each JAR. Developers could then selectively replay those changes within their own IDE to update source code. Such a hybrid approach would offer several advantages. There is no reliance on library designers to annotate API changes and specify replacements since both tasks would be carried out automatically, which also means that whether or not library designers use refactoring tools to enact changes would be irrelevant. A common formal change specification would allow any developer to update their code regardless of the IDE they used. Since the change specification would be created by one tool and read by another, developers would be spared from having to learn any new formal languages. Lastly, the developer would selectively determine which updates to apply and which to ignore. Making the developer the final decision-making authority helps offset the fact that API recommendation tool suggestions are based on heuristics and may be inaccurate. Some aspects of this hybrid approach have also been put forth by Nita and Notkin [2010].
6.5 Summary

In this chapter we addressed the limitations of our research, open issues with the Trident tool, and avenues for future work. The results of this work may be limited by learning bias in the case study and selection bias in the nature of the systems examined and in the choice of participants. We attempted to mitigate learning bias by having participants perform the case study with Trident first and repeat the task manually second. Thus the learning bias was against Trident since participants were armed with more knowledge of the task during manual refactoring. Despite this advantage, Trident—both quantitatively and qualitatively—outperformed standard tooling. Our choice of system (i.e., JaxMe) may have been simplistic but our case study showed that, even with a simple system, non-trivial migration problems arose. Our participant sample size was small and perhaps not representative of the developer population in general but both participants had numerous years of industrial experience, and Participant #1 was an expert at creating the regular expressions needed to use standard tooling yet he still faced numerous hurdles.

Trident is limited to handling API changes where a 1:1 correspondence exists between the original and replacement entity, and the time delay when processing large code bases is an open usability issue. We explained—and have partially implemented—enhancements to speed up Tridents’ performance such as: allowing developers to limit the search scope and building Trident-specific extensions atop the Eclipse performance optimizations. Handling more complex API change is left as an area for future work. Some other areas for future work include: (a) extending deprecation to account for other breaking changes, (b) integrating the Trident functionality into Eclipse as either a “quick fix” feature or directly into standard refactorings; and (c) combining API recommendation systems with refactoring references to automatically creating an API migration plan and apply it to source code.
Chapter 7

Related Work

Providing developers with assistance in adapting programs to changes in the underlying class libraries implies a need to first understand the kinds of API changes that occur as libraries evolve. Given these twin goals, our presentation of the related work, much like the research effort, is also subdivided into two parts. First a review of earlier work on collating and categorizing API changes followed by other efforts at providing tool support for such scenarios.

7.1 Related Work on API Change

Dig and Johnson [2006] provided a prominent study of API evolution. Their study relied on “manual analysis of the API changes” between two major releases of “four well known [open-source] frameworks” and one proprietary system. Later work [Dig et al., 2006b] examined a subset of the same systems using an automated approach with no appreciable difference in the findings. The specific details of the systems examined in the earlier study [Dig and Johnson, 2006] are contained in Table 7.1.

<table>
<thead>
<tr>
<th>System</th>
<th>Release A</th>
<th>Release B</th>
<th>Breaking Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eclipse</td>
<td>2.1</td>
<td>3.0</td>
<td>51</td>
</tr>
<tr>
<td>Mortgage</td>
<td>-</td>
<td>-</td>
<td>11</td>
</tr>
<tr>
<td>Struts</td>
<td>1.1</td>
<td>1.2.4</td>
<td>136</td>
</tr>
<tr>
<td>log4j</td>
<td>1.2</td>
<td>1.3 (alpha6)</td>
<td>38</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>4.0</td>
<td>5.0</td>
<td>58</td>
</tr>
</tbody>
</table>

Table 7.1: Systems studied for API change by Dig and Johnson [2006].

For all 5 systems, qualitative determination was made to classify the differences between versions which was followed by quantitative analysis of the frequency of each change. At the
first level of classification API changes were differentiated as being a non-breaking change or a breaking change. Dig and Johnson defined “a non-breaking change [as] backwards compatible” while a “breaking change is not backwards compatible”. The breaking changes were further divided into changes that “preserve behaviour … but might cause applications to fail to compile” (structural changes) versus changes where “the application might compile fine but behave differently at runtime” (behavioural transformations). A detailed list of the 20 different kinds of breaking changes (e.g., moved field, deleted method) and the 4 kinds of non-breaking changes was also provided.

The results of the Dig and Johnson study show that (a) breaking API changes occurred in all pair-wise comparisons and (b) the majority (81%–94%) of these breaking changes are behaviour-preserving structural transformations (i.e., refactorings) and, as such, they argue that there is a need for (c) a “refactoring-based” migration tool.

Our work also argues for (and implements and evaluates) a refactoring-based migration tool. However, in support of this view, we provide a study of API changes that is broader in scope and more fine-grained in its analysis. We examined API changes across 54 versions of three libraries while Dig and Johnson compare 10 versions of 5 libraries. They compared only changes to public source entities “since those entities are meant to be used by application programmers”. Whereas we compared changes to both public and protected entities since many libraries require application programmers to subclass parts of the API to gain access to functionality. These discrepancies are a symptom of our a larger differences regarding the definition of what constitutes a breaking API change. According to Dig and Johnson an API “means any component that is supposed [our emphasis] to be reused by the client and thus expected to be stable”. Thus, changes to public methods not listed in the API documentation, such as Eclipse ‘internal’ packages, are not considered breaking changes. Empirical evidence suggests that all public (and protected) methods are fair targets for application programmers thus failure to consider changes to such entities is likely to lead to underreporting of breaking
API changes. Consequently, we make no such distinctions regarding API changes; choosing to rely instead on the much stricter notion of binary incompatibilities.

Similar work by Xing and Stroulia [2006] examined and classified the change that occurred in 6 selected version of the Eclipse JDT (Java Development Toolkit) between January 27, 2002 and January 27, 2005. More specifically they undertook “pair-wise comparison [of] releases 2.0 and 2.1, 2.1.3 and 3.0, 3.0.2 and 3.1”.

The authors created the JDEvAn (“Java Design Evolution and Analysis”) tool [Xing and Stroulia, 2005a] to “recover a model of the subject system class design”. Once the structural details of two successive system versions had been extracted, they created and applied a second tool—UMLDiff—which determines if entities “in the corresponding model represent the ‘same’ conceptual entity, based on their name similarity and structure similarity”. Xing and Stroulia concede that they found “similar results” to the Dig and Johnson study. Namely that (a) 70% of the changes in the Eclipse JDT can be attributed to refactorings, and they argued that (b) for 60% of those changes a refactoring-migration tool would be useful.

In some respects, the work of Xing and Stroulia complements ours but crucial differences remain. The most immediate distinction is the sample size used in each study; we examined 54 versions of 3 libraries while their study used 6 versions of single system. The choice to study a single systems is justified with the claim that Eclipse has an “intensive and varied structural-evolution history” and, thus, is an ideal vantage point from which to assess “how much of this evolution involves refactorings”.

We contend that studying more systems and more versions bolsters the generalizability of our results and there are several reasons why the same cannot be said for a study of just Eclipse. First, other researchers [Murphy-Hill et al., 2009] have shown that refactoring usage is less amongst refactoring designers than it is among regular users. This is especially pertinent because Xing and Stroulia examined only the Eclipse JDT subproject which is home to the Eclipse refactoring tools and presumably maintained by those same refactoring designers.
Consequently, their results on the extent of API changes between library versions may under (or over) report change.

Second, the authors “excluded from [their] analysis about 75% of all the changes that according to [their] understand[ing] . . . represent the introduction of new features or the removal of [an] obsolete API”. Conversely, we include all changes, including alterations to obsolete APIs, in our reporting of binary incompatibilities. Since developers who were reliant on those obsolete APIs will need to update their code to the newer versions, we feel our more inclusive approach better captures the full range of problems that developers will encounter during library migration.

Third, differences in what code is excluded (or included) in our respective analyses is indicative of a larger conceptual difference in the focus of each study. Xing and Stroulia were interested in “what fraction of code modifications are refactorings” while our emphasis is on what fraction of library changes give rise to binary incompatibilities. Both bodies of work attempt to categorize the changes that occur, with Xing and Stroulia using UMLDiff to automatically classify changes while we manually inspected the code base and Javadoc. As a result Xing and Stroulia consider (a) types of changes that we ignore (e.g., addition of new entities) they also examine (b) a greater number of changes and (c) posit a refactoring reason for each change. Using manual classification means, we (a) examine fewer changes but (b) rely on expert opinion to classify each change which lends a different tenor to the data and (c) make no claims as to why the change occurred.

The research efforts chronicled above are the latest contributions in a sustained attempt to understand and track why source code changes. Godfrey and Zou [2005] used origin analysis to “aid in the detection of merging and splitting of files and functions in procedural code”. This technique uses several criteria (i.e., name, afferent/efferent method calls) to match code elements. A developer can then adjust the weighting given to each criterion and select a match from amongst the several search candidates. Kim et al. [2005] built upon Godfrey and Zou
[2005] work by providing “significance analysis” to determine the prediction value of each criterion, individually or in combination, on a project-specific basis. Similarly Taneja et al. [2007] extend their earlier work on Refactoring Crawler [Dig et al., 2006a] to create RefacLib which, like Kim et al. [2005], uses weighted average of eight heuristics to find evidence of refactoring between library versions; though the approach is different, the systems studies are identical to the study by Dig and Johnson [2006].

If you happen to have access to the library source code repository—and assuming the repository itself has not evolved, which happened in many of the projects we reviewed for this work—then Weissgerber and Diehl [2006] proposed mining the repository history to detect refactorings from classes involved in the same commit operation. Xing and Stroulia [2004] also leveraged repository access to show class co-evolution where “clusters of classes change in ‘parallel’ ways”. Lastly, Kim et al. [2007] extract the change details from a repository “as first-order relational logic rules” which, they argue, provides a more structured presentation of changes that could be reused by other software engineering tools.

Work outside the Java language realm by Neamtiu et al. [2005] looked at comparing “the source code of different versions of a C program”. However, they began with the premise that “function names are relatively stable over time” and examined only the internal details of methods that shared the same name, which is a considerably narrower scope then the changes we consider.

7.2 Related work on API Migration Tool support

Previous work that is conceivably relevant to the problem of library migration can be classified as lexically-based, syntactically-based, and semantically-based.

Lexically-based approaches consist of standard search-and-replace features present in IDEs and in traditional search tools such as the grep family of Unix tools (e.g., [Wu and Manber,
Such tools can (a) fail to help with the transformation of located references and/or (b) demand precision from the developer in specifying minor lexical details, resulting in either false positives or false negatives. As we have seen in our case studies, lexical tools fall short of our approach.

Syntactically-based approaches add knowledge of syntactic structures into the mix, thus enabling discrimination of references from declarations. For example, TXL [Cordy, 2006] is a syntactic transformation language that one could conceivably use to locate references and refactor them; however, we have seen in our case studies the utility of also leveraging semantics to locate only references of certain types.

Traditional approaches to program transformation (e.g., [Feather, 1989]) demand the presence of formal specifications of the source code, which tend to be absent in industrial settings. Semantic grep [Bull et al., 2002] suffers from being burdensome on the developer just as with grep.

7.2.1 Semantically-Based tools

More sophisticated efforts at creating API migration tools fall into three categories: (1) tools that require formal API transformation specifications to be written (either by the library creator or the library user) or (2) tools that rely on capturing information about library changes as they occur or (3) tools which create custom code to bridge the differences between API’s.

7.2.2 Migration Tools: Formal Change Specifications

The first category contains contributions from quite a number of researchers. One of the earliest contributions was from Chow and Notkin [Chow and Notkin, 1994] who required “library maintainer[s] to annotate change functions with rules that are [then] used to generate tools that will update the applications that use the updated libraries”. Not only is this approach limited to functions but it also requires “full source-code for the library”. A condition which is impracti-
cal, if not impossible, when dealing with commercial libraries as is the case at Chartwell Technology. Moreover, it is unrealistic to expect library maintainers to annotate changed functions, never mind specifying change rules, given that our study of APIs shows that for the vast majority of library changes no advance warning is given through the use of the @deprecation tags. Of the few changes that are shown as deprecated ahead of time, even fewer use the @see tag to indicate a replacement.

Analogous research by Tip et al. [2003], Balaban et al. [2005], and Nita and Notkin [2010] requires the developer using the library to write a specification of the transformation to apply in migrating from one library to the next.

Tip et al. require that a complete transformation specification be written out using a complex notation involving type constraints. This formalism is “designed to efficiently compute the maximal set of allowable source-code modifications” and is not intended to be an interactive refactoring. Moreover their approach needs “access to a program’s [i.e., both library and client code] full source” which we have already shown is an impractical requirement. A need for complete access to the source code also implies this approach does not deal with declaratively incomplete code as is the case with dangling references.

The work of Balaban et al. aims to build upon that of Tip et al. The focus of this work is on migrating from legacy classes to their replacements. However, the formalism of type constraints is maintained with all its accompanying shortcomings. The advantage of a stricter specification is the ability to preserve type correctness and guaranteeing synchronization safety whereas we offer no such guarantee, electing instead to allow the developer to be the arbiter of what modifications are sensible. Forgoing type correctness enables Trident to offer novel refactorings (e.g., move method to a different type) that are not possible otherwise.

Recent research by Chow and Notkin presents a technique called twinning. As they describe it, twinning involves creating “a mapping [their emphasis] that transition[s] a program from using one API to using an alternative API”. As with the previous two approaches, a de-
veloper must learn the syntax of a mapping language, albeit simpler than the type constraint language, to specify how different versions of a library correspond to each other. The authors argue that “writing a mapping . . . can imaginably have other benefits such as helping the developer crystallize the intended relationship between the [program] variants“. Developer behaviour observed in our case study suggests the opposite is true; developers ‘crystallize’ their understanding of differences between API’s by following an iterative, example-based process of discovery. Participant #2 stated he preferred “working through the list of [compilation] errors incrementally until the change was complete”.

Furthermore, our incremental approach has the advantage that the developer need only deal with those differences between API’s that produce errors. As noted in the motivating example the errors at Chartwell Technology arose from 7 of the 167 differences between JDOM versions. A complete mapping of all 167 program ‘variants’ would have been excessive in this case.

Finally, twinning is best suited for API transformations that “involve fully removing all the uses of a type replacing it with another type”. Our study of API changes, motivating example and case study show that such complete type replacements are outnumbered by partial type replacements and modifications to methods which Trident is optimized to handle.

In general there are 3 problems with the aforementioned approaches: (1) they are not extensible to general, many-to-many transformations, yet are intended to be automated; (2) they tend to focus on correctness and precision rather than getting the worst of the job done for the developer; and (3) writing out transformation specifications is not how developers think about performing library migrations [Boshernitsan et al., 2007] As Murphy-Hill says [Murphy-Hill, 2009], “programmers sometimes want to break code [temporarily] ... and ... programmers already know how to fix compilation errors, so having them fix compilation errors should be easier than fixing unfamiliar refactoring tool errors.”
7.2.3 Migration Tools: Change Recording and Replay

The second category of API migration tools emphasizes automatically recording library changes as they occur and utilizing that information to update client code. The Spyware tool [Robbes and Lanza, 2008] is an IDE plugin that “tracks the changes that a developer performs on a program as they happen”. Spyware does not work on declaratively incomplete code and it lacks a mechanism to convey the recorded changes from library maintainer to application developer.

CatchUp! [Henkel and Diwan, 2005] provides a means to record automated refactorings and share that information with developers so that the changes can be replayed to update client code; this idea has been incorporated as refactoring scripts in industrial IDEs. Unfortunately, CatchUp! restricts both the library maintainer and application developer to using the same IDE. As well, it requires that the library changes be performed with automated refactorings. Changes arising from manual edits or other means are ignored. This is problematic since 20% of API changes [Dig and Johnson, 2006, Xing and Stroulia, 2006] are not refactoring related. Recording of refactorings is now included with the standard Eclipse distribution at the “Create Refactoring Script” option and playback is supported via the “Migrate JAR” feature.

7.2.4 Migration Tools: Custom Adaptation Code

The third category of tools creates specialized code that is designed to bridge the differences between APIs. Proposals have been put forth to: (a) forward calls from a deprecated method to its replacement [Perkins, 2005] or (b) leverage recordings of changes to library code to automatically create adapter classes [Savga and Rudolf, 2007, Savga et al., 2008, Dig et al., 2008] or (c) infer changes that have occurred in a set of files to generate adapter code [Andersen and Lawall, 2008, Tansey and Tilevich, 2008]. Such tools are aimed at library developers, who have access to the repositories containing the history of their librarys changes, but cannot modify clients who rely on functionality they provide. These techniques are aimed at legacy systems, however, where adapting the source code using the library is impractical or undesirable.
For example, Andersen and Lawall [2008] describe an approach that mines a patch repository for common transformations that respond to interface changes. Their approach is geared towards operating systems and device drivers; a repository of patches is unlikely to exist in the case of libraries which are often simply released as successive versions due to their smaller size. Also, they give an example in which a function call has had an argument eliminated, and for which their technique is incapable of determining a complete transformation from the original call to the new one. This is exactly the kind of case where manual intervention is most needed, and which we aim to address. The work of Tansey and Tilevich [2008] focuses on the more limited problem of refactoring annotations. Their machine learning technique “infers a concise set of semantics-preserving transformation rules from two versions of a single class” but requires that these exemplar classes be manual adapted by the developer. Moreover as the number of changes increases so does the number of exemplars’ required, which could be a more costly technique overall than manual intervention, and it still does not provide complete coverage.

Ultimately the creation of intermediate code as a bridge between library versions is a short term compromise that delays, but does not eliminate, the problems of API migration. Intermediate adapter code (1) adds performance overhead during runtime and once again places the onus on the library maintainer to provide a solution (i.e. providing access to library source code or providing adapter classes with each version); (2) provides only a partial solution as some kinds of library changes cannot be hidden behind an adaptive layer but need to be addressed in the client code.

7.2.5 Migration Tools: API Recommenders

Diff-CatchUp [Xing and Stroulia, 2007] and SemDiff [Dagenais and Robillard, 2008] both recommend replacements for dangling references due to library migration. Both approaches mine a source code repository (such as the library’s own implementation) to determine how
calls in other systems have been migrated. Aside from the fact that these approaches will operate only when a reliable repository exists, these approaches only make recommendations, failing to aid in the actual transformation process itself. In our case study, the problem was not what the dangling references should be replaced with, but how to actually perform the task: thus, such recommenders are complementary to our work.

7.3 Summary

Existing work does not suffice to support the refactoring of references. Standard transformation approaches can demand excessive precision from the developer in terms of the search query, can be excessively rigid about syntactic details, or can fail in the presence of broken semantics. Previous work to support library migration has focused on either: strong notions of type correctness, leading to poor performance or difficulty in specifying transformations; or solely on recommending alternative calls, rather than actually performing transformations. Our approach addresses these shortcomings.
Chapter 8

Conclusion

We have examined the problem of refactoring references, particularly with respect to the library migration problem, and demonstrated that it is not well-solved with state-of-the-practice approaches. The experiences of our industrial partner, Chartwell Technology, show that libraries are extensively used in industrial systems and can be expected to change. Even minor API changes can have considerable consequences and impact in terms of both time and money. To better understand library evolution, we examined many versions of a few industrially-relevant software systems and found that API changes can be frequent, without warning, and severe.

The average number of changes for each transition in the libraries we studied is between 13 and 28. In addition we noted that the majority of change events involved the removal of entities, and approximately half of the change events are related specifically to methods. Further compounding the problem, we found that most of these changes were not accompanied with documentation about the recommended migration path.

To address this problem we have created a lightweight approach for exemplar-based search-and-replace combining lexical, syntactic, and semantic criteria for selection and refactoring as a developer engaged in a library migration task sees fit. Motivated by the work of Henkel and Diwan [2005] and Balaban et al. [2005] we implemented a tool that: (1) allowed the developer to operate independently of the library designer, (2) offered functionality that was like standard refactorings (e.g., preview and undo) but was also (3) unlike standard refactorings in the types of changes it would permit (e.g., replace a method expression with an arbitrary expression).

Our case studies involved two industrial developers who undertook a restricted library migration task both with and without our approach, as embodied in the Trident tool. Despite significant biases towards the status quo, Trident was seen as being of significant benefit in
terms of time to complete the task and in aiding the developer’s comprehension. In partic-
ular we found that developers struggled to create regular expressions that properly matched
the changes they intended to make. The developers also struggled with copy-and-paste errors,
coupled with an inability to undo changes globally that forced them to resort to making a con-
siderable number of changes by hand. Conversely, when provided with Trident tool support,
developers made fewer errors, took a shorter period of time, undertook fewer manual modifi-
cations, and appreciated the ability to undo and redo changes iteratively as their understanding
of the task progressed.

Trident is limited to handling API changes where a 1:1 correspondence exists between the
original and replacement reference, and the time delay when processing large code bases is
an open usability issue. We have made some progress at solving the usability issue, but leave
handling more complex API changes as an area for future work. Our results may be limited by
learning bias in the case study and selection bias in the nature of the systems examined and in
the choice of participants. However, we have attempted to mitigate the effects of bias by either
ensuring the bias was against our research or have attempted to show how the bias does not
detract from the results.

We intend to continue our work to expand the range of scenarios in which our approach
applies, to improve the usability of our tool support, and to evaluate it with increasing formality.
The results presented herein are a strong indication of the value of continued investment in this
line of research.

8.1 Contributions

The contributions of this work are:

- a study of binary incompatible API change in the wild and
- the construction and evaluation of a tool that allows source code to adapt to such changes
by refactoring references.

8.2 Future Work

We proposed refactoring of references as a solution to the problem of API evolution but there remain ample areas for further research. Dangling references also arise in test-driven development, pragmatic reuse, and practical development scenarios; though we have mentioned examples from these domains, further study is required to fully understand the needs of developers in such instances and adapt Trident to accommodate them. Even within the area of API evolution, a larger, controlled study using Trident would bolster the generalizations of our findings and, potentially, yield insights for new work. Such a study should also examine more applications and more libraries to see how the specific usage of libraries within applications affects their ability to respond to change.

Combining multiple research approaches to dealing with API change is another promising area of investigation. For example, it should be possible to (a) use binary incompatibility to locate breaking changes and then (b) input those changes into API recommendation systems to generate replacement recommendations which (c) could be written out using a formal specification language that (d) could be read by tools within each IDE to alter references and update client code. Any efforts that can intelligently automate the process of dealing with software change would be a boon.
## Glossary

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deprecated</td>
<td>See deprecation definition.</td>
</tr>
<tr>
<td>Deprecation</td>
<td>A program element is marked as deprecated to discourage programmers from using it, typically because it is dangerous, or because a better alternative exists.</td>
</tr>
<tr>
<td>Eclipse</td>
<td>Popular and widely used, open-source multi-language software development environment.</td>
</tr>
<tr>
<td>Grep</td>
<td>Text search program originally written for Unix and adapted to other platforms. It relies on users defining regular expressions to locate text patterns.</td>
</tr>
<tr>
<td>HTMLUnit</td>
<td>A Java library for unit testing Web based applications.</td>
</tr>
<tr>
<td>JDOM</td>
<td>A Java library for accessing, manipulating, and outputting XML data from Java code.</td>
</tr>
<tr>
<td>log4j</td>
<td>A Java library for printing log output to different local and remote destinations.</td>
</tr>
<tr>
<td><strong>Notation</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Refactoring</td>
<td>Refers to the process of changing the structure of a program—to improve quality attributes such as understandability, maintainability, extensibility, etc.—without changing its behaviour.</td>
</tr>
<tr>
<td>Trident</td>
<td>A prototype research tool created for this thesis.</td>
</tr>
</tbody>
</table>
Bibliography


Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Design*. Addison-Wesley, 1994.


Michael W. Godfrey and Lijie Zou. Using origin analysis to detect merging and splitting of


Appendix A

Ethical Review and Informed Consent

This research has been conducted as a portion of a larger research program, entitled “Pragmatic Reuse” (File #5842), and given ethics approval thereunder on 11 November 2008, renewed 27 October 2009 as valid until 30 April 2013. The ethics review application form that covers this research can be found on pages 116–126. The consent form given to participants can be found on pages 127–129.
CONJOINT FACULTIES RESEARCH ETHICS BOARD
APPLICATION FOR ETHICS REVIEW
Research Services, ERRB Building, Research Park

Be sure to consult the “Instructions to Applicants” when completing this form.

Copies: Faculty (and students from those Faculties/Departments which do not have their own Ethics Committees): Submit 1 original and 1 photocopy including all supporting documentation to Research Services, ERRB Building, Research Park.

Copies: Students – Variable*: Submit the original and the number of copies required by your Faculty/Department Ethics Committee.

* See Ethics website for list of Committee Chairs and specific locations for submission of applications.

CFREB Ethics Certification extends only to those individuals who have a current University of Calgary affiliation (student, faculty, staff). For the purposes of this application, “applicant and co-applicant” refer to those individuals who are applying for ethical clearance from the University of Calgary. This may be different from the person who is listed as the Principal Investigator/Co-investigator on the project.

1.1 Applicant:

<table>
<thead>
<tr>
<th>Family Name</th>
<th>Given Name and Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walker</td>
<td>Robert J.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department/Faculty</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mailing Address (complete only if different from Department/Faculty)</th>
<th>E-mail Address</th>
<th>Telephone (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="mailto:walker@ucalgary.ca">walker@ucalgary.ca</a></td>
<td>210-9593</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title/Position (Check One)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[X] Full-time Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Adjunct Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Postdoctoral Fellow</td>
<td></td>
</tr>
<tr>
<td>[ ] Sessional Instructor</td>
<td></td>
</tr>
<tr>
<td>[ ] Professor Emeritus</td>
<td></td>
</tr>
<tr>
<td>[ ] Staff Member</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Graduate Student:</th>
<th>Master's</th>
<th>Ph. D</th>
<th>Other (please specify):</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Undergraduate Student</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other (please specify):</th>
<th></th>
</tr>
</thead>
</table>

1.2 Supervisor, if applicable:

<table>
<thead>
<tr>
<th>Family Name</th>
<th>Given Name and Initial</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Department/Faculty</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Mailing Address (complete only if different from Department/Faculty)</th>
<th>E-mail Address</th>
<th>Telephone (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 1.3 Co-Applicant, if applicable:

<table>
<thead>
<tr>
<th>Family Name</th>
<th>Given Name and Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cossette</td>
<td>Bradley E.</td>
</tr>
<tr>
<td>Cottrell</td>
<td>Rylan</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department/Faculty</th>
<th>E-mail Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science/Science</td>
<td><a href="mailto:cossette@cpsc.ucalgary.ca">cossette@cpsc.ucalgary.ca</a></td>
</tr>
<tr>
<td></td>
<td><a href="mailto:cottrell@cpsc.ucalgary.ca">cottrell@cpsc.ucalgary.ca</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title/Position (Check One)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] Full-time Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Adjunct Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Postdoctoral Fellow</td>
<td></td>
</tr>
<tr>
<td>[ ] Staff Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Sessional Instructor</td>
<td></td>
</tr>
<tr>
<td>[ ] Professor Emeritus</td>
<td></td>
</tr>
<tr>
<td>[ ] Graduate Student: [ ] Master's [X] Ph. D [ ] Other (please specify):</td>
<td></td>
</tr>
<tr>
<td>[ ] Undergraduate Student</td>
<td></td>
</tr>
<tr>
<td>[ ] Other (please specify):</td>
<td></td>
</tr>
</tbody>
</table>
### 1.3 Co-Applicant, if applicable:

<table>
<thead>
<tr>
<th>Family Name</th>
<th>Given Name and Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kapur</td>
<td>Puneet</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department/Faculty</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science/Science</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mailing Address (complete only if different from Department/Faculty)</th>
<th>E-mail Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="mailto:puneet@kapur.org">puneet@kapur.org</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Telephone (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>403-220-5017</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title/Position (Check One)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] Full-time Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Adjunct Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Postdoctoral Fellow</td>
<td></td>
</tr>
<tr>
<td>[ ] Staff Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Sessional Instructor</td>
<td></td>
</tr>
<tr>
<td>[ ] Professor Emeritus</td>
<td></td>
</tr>
<tr>
<td>[ ] Graduate Student:</td>
<td>[ ] Master's</td>
</tr>
<tr>
<td>[ ]</td>
<td>[ ] Ph. D</td>
</tr>
<tr>
<td>[ ]</td>
<td>[ ] Other (please specify):</td>
</tr>
</tbody>
</table>

| 1.3 Co-Applicant, if applicable: |

<table>
<thead>
<tr>
<th>Family Name</th>
<th>Given Name and Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makady</td>
<td>Soha</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Department/Faculty</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science/Science</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mailing Address (complete only if different from Department/Faculty)</th>
<th>E-mail Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><a href="mailto:shmmakad@ucalgary.ca">shmmakad@ucalgary.ca</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Telephone (local)</th>
</tr>
</thead>
<tbody>
<tr>
<td>403-220-5017</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Title/Position (Check One)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] Full-time Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Adjunct Faculty Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Postdoctoral Fellow</td>
<td></td>
</tr>
<tr>
<td>[ ] Staff Member</td>
<td></td>
</tr>
<tr>
<td>[ ] Sessional Instructor</td>
<td></td>
</tr>
<tr>
<td>[ ] Professor Emeritus</td>
<td></td>
</tr>
<tr>
<td>[ ] Graduate Student:</td>
<td>[ ] Master's</td>
</tr>
<tr>
<td>[ ]</td>
<td>[ ] Ph. D</td>
</tr>
<tr>
<td>[ ]</td>
<td>[ ] Other (please specify):</td>
</tr>
</tbody>
</table>

### 1.4 Additional Research Team Members: Provide as an attachment.

If other person or persons is/are involved in the project, but not affiliated with the University of Calgary, please provide his or her name, organization/employer, affiliation and other details to identify them.
## 2. Project Details:

<table>
<thead>
<tr>
<th>2.1 Exact Title of the Project</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pragmatic Reuse</td>
</tr>
</tbody>
</table>

| 2.2 Is this an amendment/modification to a previously approved protocol? [x] No [ ] Yes (Note: see Information to Help Applicants for more details. Separate procedures apply when modifications do not involve significant changes to the original protocol. Please contact the CFREB office [220-3782] if you are unsure whether the changes to an existing protocol constitute a modification/amendment, or are significant enough to warrant a new application.) |

<table>
<thead>
<tr>
<th>2.3 Status of funding/support for the project - please choose one:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ ] Unfunded project   [ ] Funding pending   [x] Funding received</td>
</tr>
</tbody>
</table>

Sponsor(s)/funding agency(s): [ ] SSHRC [x] NSERC [ ] CIHR Other (please specify):

Name of investigator(s) applying for or receiving funding: Robert J. Walker

Project title as submitted to funding agency (if different than title of ethics submission): Unanticipated Reuse

<table>
<thead>
<tr>
<th>2.4 Anticipated start date of work involving human participants (mm/yy)</th>
<th>Anticipated completion date of research activity; for graduate thesis or dissertation, please list anticipated date of defense (mm/yy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01/09</td>
<td>04/13</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.5 List the location(s) where the data will be collected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Department of Computer Science, in an office to be determined or in the Software Engineering Laboratory</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>2.6 Are other approvals/permissions required where this research will occur? [x] No [ ] Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>If yes, provide a copy of the approval: [ ] Attached [ ] To follow (Specify where from):</td>
</tr>
</tbody>
</table>
2.7 Provide a succinct summary of the purpose, objectives, and aims of the research. Describe your methodology, and what will be required of the human participants. Please use language that can be understood by a non-specialist. Up to 1 additional page may be added, if required. (Note: Project descriptions exceeding the two-page limit will not be considered.) REMINDER: Be sure to include a copy of any questionnaire(s) or test instrument(s).

Reusing source code in a manner for which it has not been designed (which we term unanticipated reuse) is traditionally regarded as poor practice. While this belief is widely held, it has been little studied in the literature. We seek to determine how unanticipated reuse is performed in practice, what effects it has on software quality (relative to the traditional alternatives), and new techniques to help developers perform it better.

The project will involve five experiments. The student involved with each is indicated in parentheses.

Experiment 1: Polylingual Dependency Analysis in Pragmatic Reuse Tasks (Cossette)
Experiment 2: Test Case Selection and Transformation in Pragmatic Reuse Tasks (Makady)
Experiment 3: Semi-automated Small-scale Pragmatic Reuse (Cottrell)
Experiment 4: API Navigation Relative to the Developer’s Context (Kapur)
Experiment 5: Semi-automated Boundary Determination in Pragmatic Reuse Tasks (Cottrell)

The experiments will vary in the details of the technical approach being studied, but the protocol will be the same for all. Each experiment will seek to determine the relative efficacy of alternative techniques, a novel one created by us and a control that is either current in industrial practice or a technique that we have previously created and/or studied.

Each participant will first be asked to fill out a questionnaire (see “pre-study questionnaire”) on their development background and experience with the techniques under study. Each participant will then be asked to complete two small training tasks to familiarize them with our tool support and the control. No time limit will be imposed on the training phase, though the expectation is that this should take 10-20 minutes; the participant can ask any question whatsoever during the training phase and receive a completely candid response.

Each participant will be given two predetermined software development tasks to perform, and they will be asked to use a particular technique in performing each; our novel technique will be used for one task and a control technique will be used for the other. A time limit will be imposed (45 minutes) for each task, for the sake of minimizing the physical and mental effort required. Specific criteria for the completion of each task will be defined (such as the successful execution of a provided test suite), and these will be clearly explained to the participant prior to the start of the task. The tasks can be abandoned prior to success and prior to the time limit if the participant so chooses. A short questionnaire about the task (see “post-task questionnaire”) will be given before the next is started.

After the tasks are done, a questionnaire (see “post-study questionnaire”) will be given about the participant’s impressions of the relative merits of the approaches and how ours might be improved. Participants will be compensated $20 for their time at the conclusion of the experiment.
2.7 Provide a succinct summary of the purpose, objectives, and aims of the research. Describe your methodology, and what will be required of the human participants. Please use language that can be understood by a non-specialist. Up to 1 additional page may be added, if required. (Note: Project descriptions exceeding the two-page limit will not be considered.) REMINDER: Be sure to include a copy of any questionnaire(s) or test instrument(s).

Reusing source code in a manner for which it has not been designed (which we term unanticipated reuse) is traditionally regarded as poor practice. While this belief is widely held, it has been little studied in the literature. We seek to determine how unanticipated reuse is performed in practice, what effects it has on software quality (relative to the traditional alternatives), and new techniques to help developers perform it better.

The project will involve five experiments. The student involved with each is indicated in parentheses.

Experiment 1: Polylingual Dependency Analysis in Pragmatic Reuse Tasks (Cossette)
Experiment 2: Test Case Selection and Transformation in Pragmatic Reuse Tasks (Makady)
Experiment 3: Semi-automated Small-scale Pragmatic Reuse (Cottrell)
Experiment 4: API Navigation Relative to the Developer’s Context (Kapur)
Experiment 5: Semi-automated Boundary Determination in Pragmatic Reuse Tasks (Cottrell)

The experiments will vary in the details of the technical approach being studied, but the protocol will be the same for all. Each experiment will seek to determine the relative efficacy of alternative techniques, a novel one created by us and a control that is either current in industrial practice or a technique that we have previously created and/or studied.

Each participant will first be asked to fill out a questionnaire (see “pre-study questionnaire”) on their development background and experience with the techniques under study. Each participant will then be asked to complete two small training tasks to familiarize them with our tool support and the control. No time limit will be imposed on the training phase, though the expectation is that this should take 10-20 minutes; the participant can ask any question whatsoever during the training phase and receive a completely candid response.

Each participant will be given two predetermined software development tasks to perform, and they will be asked to use a particular technique in performing each; our novel technique will be used for one task and a control technique will be used for the other. A time limit will be imposed (45 minutes) for each task, for the sake of minimizing the physical and mental effort required. Specific criteria for the completion of each task will be defined (such as the successful execution of a provided test suite), and these will be clearly explained to the participant prior to the start of the task. The tasks can be abandoned prior to success and prior to the time limit if the participant so chooses. A short questionnaire about the task (see “post-task questionnaire”) will be given before the next is started.

After the tasks are done, a questionnaire (see “post-study questionnaire”) will be given about the participant’s impressions of the relative merits of the approaches and how ours might be improved. Participants will be compensated $20 for their time at the conclusion of the experiment.
4. Informed Consent

4.1 Described the informed consent process. Provide a copy of your consent form. If there is no written consent form, please provide an explanation for this and details about your alternative procedures. If obtaining verbal consent, a script containing the same points normally covered by written consent is required. Are participants minors or, for other reasons, not able to provide fully informed consent? Explain and justify, and describe alternative procedures (e.g. parental consent).

The written consent forms are attached to this application. They are identical save for the contact information provided.

4.2 When and how will people be informed of the right to withdraw from the study? What procedures will be followed for people who wish to withdraw at any point during the study? What happens to the information contributed to this point? Please note that the CFREB does not require that researchers withdraw/destroy partial data in cases of participant withdrawal, provided that it is made clear on the informed consent form that data collected to the point of withdrawal will be retained/used.

Participants will be informed of the right to withdraw from the study prior to the beginning of their involvement when they are asked to sign the consent form. People who wish to withdraw from the study will be allowed to do so at any time, but the information they have contributed to that point may be retained.

4.3 Do you plan follow-up procedures with participants? [x] No [ ] Yes, if yes, what are they? Does your research design require formal debriefing? [x] No [ ] Yes, if yes, please provide details about the procedures you will use.

5. Privacy: Confidentiality and Anonymity:

5.1 Check all that apply: Participant contributions will be: [ ] public and cited; [x] anonymous; [ ] confidential. Explain the steps you propose to respect an individual’s privacy. Describe these precautions in terms of access to raw data, as well as in terms of the write-up of the results. For example, will data be reported in aggregate? Will participants select a pseudonym? Will participants be asked to review their contribution before inclusion? (Please note that the CFREB does not require that participants be given the option of reviewing their data, provided they are aware that this opportunity will not be offered to them. Should you wish to provide participants with a chance to review material attributed to them, it is recommended that you set a specific time limit [e.g. within two weeks of receiving the material] by which participants must contact you with any suggested changes to material attributed to them, with a lack of response within that time indicating that the participant approves of the material as is, in order to avoid delays to your research. This timeline should be made clear in the consent protocol.) Who gets the data and in what form?

The participant’s name will be collected, but it will not be publicly reported (i.e., a pseudonym such as “Participant 1” will be assigned to each). No further effort will be made to anonymize the results. Results and contributions (including participant comments, participant responses, indications of participant performance, and the output of tasks) may be publicly reported and labelled with the aforementioned pseudonyms. Participants will not be audio- or video-taped for these studies.
5.2 Provide specific details about the security procedures for the data as well as plans for the ultimate disposal of records/data. Who will have access to confidential data now or in the future? Specify the length of time the data will be retained and the plans for disposal of records/data. (Note: The CFREB does not have specific data retention or destruction requirements. Researchers are free to retain data for long periods of time, or archive data indefinitely, provided this is made clear to participants in the informed consent protocol, and continued/future use of the data is consistent with what is described by the researcher[s] within this application.)

Physical copies of raw data will be stored in a locked drawer or cabinet in my office, and raw data stored on a computer will be in password-restricted locations. This raw data will be accessible only to me and to the particular student involved in the respective, individual experiment, and will be archived for a term not to exceed five years, for possible future use.

6. Estimation of Risks: Will this study involve the following? Please check Y when responding, see also Section 3– Information to Help Applicants

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>Minimal Risk</th>
<th>More than Minimal risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Psychological or emotional manipulations – might a participant feel demeaned, embarrassed, worried or upset? Could subjects feel fatigued or stressed?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.2 Are there questions that may be upsetting to the respondent?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.3 Does your study have the potential for identifying distressed individuals?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.4 Is there any physical risk or physiological manipulation?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.5 Is any deception involved? Withholding of information from, or misinforming, participants?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.6 Is there any social risk - possible loss of status, privacy and/or reputation?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.7 Do you see any chance that subjects might be harmed in any way?</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.8 Is there any potential for the perception of coercion? That is, might prospective participants feel pressured to participate in the research (due to, for instance, actual or perceived power relationships between those involved in recruiting and those being recruited, e.g. manager/employee or teacher/student)?</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.9 Are the risks similar to those encountered by the subjects in everyday life?</td>
<td>[x]  Yes</td>
<td>[ ] No</td>
<td>if “no”, elaborate</td>
</tr>
</tbody>
</table>
• If you answered, “more than minimal risk” to any of the above, describe the manipulations and/or potential risks as well as the safeguards or procedures you have in place. Please provide justification for any risks involved and explain why alternative approaches involving less risk cannot be used. Use additional pages, as required.

N/A

• If your study has the potential to upset or distress individuals, arrangements must be made to mitigate such effects. Describe the arrangements you have made. Have participants been informed of any costs to be incurred by them for services? See “Provision for Rescue – Guidelines for Applicants”

N/A

• If your study has the potential to identify upset or distressed individuals, you must describe the arrangements you have made (if any) to assist these individuals. If you do not make any arrangements, please explain why. Have participants been informed of any costs to be incurred by them for services?

N/A

• If, prior to the start of the research session, participants will not be fully informed of everything that will be required of them or deliberately misinformed about some aspect of the study, explain why. Please describe the procedures in detail and justify why deception is necessary to conduct the research.

N/A

• If the potential for any perception of coercion exists, please explain what measures have been put in place to minimize the possibility that individuals will feel pressured to participate.

For student subjects: N/A, as no power relationships will exist between me and the potential participants.

For industrial subjects: N/A.
### 7. Benefits

- What are the likely benefits of the research to the researcher, the participants, the research community and society, at large, that would justify asking people to participate?

The researcher may benefit by gaining an understanding of the usefulness of the tools/approaches under investigation. This will aid in the further development of tools to assist software developers.

Participants may benefit by helping to further research in tools/approaches that are designed to assist software developers such as the participants themselves.

The research community may benefit through gaining additional knowledge regarding new tools/approaches to aid software development.

Society at large may benefit through improved efficiencies in the software development process, which may improve software quality while lowering cost.
8. Signatures

I/We, the undersigned, certify that (a) the information contained in this application is accurate; (b) conduct of the proposed research will not commence until ethical certification has been granted; (c) the Board will be advised of any revisions to the protocol arising before or after ethical certification is granted; (d) an annual renewal report will be filed 12 months from the date that ethics approval is issued, and a final report will be filed immediately upon completion of research activity. Failure to submit renewal or final reports in a timely manner will be considered a breach of University and Tri-Council policy, and may result in the suspension of research funding and/or the research being rendered academically invalid; students who fail to submit reports may be barred from graduating. Conduct of research using human subjects that has not received ethics certification is a breach of University policy on integrity in scholarly activity.

Applicant’s signature: ___________________________ Date: ______________

Co-applicant’s signature: ___________________________ Date: ______________

Co-applicant’s signature: ___________________________ Date: ______________

Co-applicant’s signature: ___________________________ Date: ______________

Co-applicant’s signature: ___________________________ Date: ______________

Supervisor’s Signature: I have been involved in the preparation of this application, and agree with the information it contains.

Supervisor’s Signature: ___________________________ Date: ______________

<table>
<thead>
<tr>
<th>PROTOCOL CHECKLIST – required</th>
<th>N/A</th>
<th>Attached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy of the verbal or written explanation that will be provided to participants before they are asked for consent to participate</td>
<td></td>
<td>(see 3.1)</td>
</tr>
<tr>
<td>Copy of the informed consent(s) that will be distributed to each participant.</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>If written consent is not used, a detailed explanation of alternative procedures is required in Section 4 of this application, along with one or more of the following:</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>• If verbal consent is to be obtained, (e.g. telephone surveys), a script containing the equivalent points covered by written consent is required.</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>• Totally anonymous online or mail out questionnaires: Signed consent is not necessary. A covering letter, containing the equivalent points covered by written consent, is required.</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Copies of questionnaire(s), sample questions or thematic overview, interview guide</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Recruitment: Your recruitment notice, advertisement, and/or information sheet as well as that used by a sponsor or supportive organization, as may be applicable</td>
<td></td>
<td>(see 3.1)</td>
</tr>
<tr>
<td>Documents or information specific to or requested by the potential sponsor.</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>Completed and signed application for review with the required number of copies.</td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

Revised: 03/07

Note: The information contained in this application is collected under the authority of the Freedom of Information and Protection of Privacy (FOIP) Act. It will be used to evaluate your application for ethics certification. Anonymized data will also be used to fulfill reporting obligations.

If you have any questions about the collection or use of this information, please contact the Ethics Resource Officer (Research Services, ERRB Building, Research Park) at (403)220-3782.
Name of Researcher, Faculty, Department, Telephone & Email:
Puneet Kapur, Department of Computer Science, pkapur@ucalgary.ca

Supervisor:
Dr. Robert J. Walker, Department of Computer Science, walker@ucalgary.ca

Title of Project:
Pragmatic Reuse

This consent form, a copy of which has been given to you, is only part of the process of informed consent. If you want more details about something mentioned here, or information not included here, you should feel free to ask. Please take the time to read this carefully and to understand any accompanying information.

The University of Calgary Conjoint Faculties Research Ethics Board has approved this research study.

Purpose of the Experiment:
Developers often rely on third-party code libraries (JAR files, DLL files) as a resource when developing their own software. For instance the java.util libraries provide high quality reusable data structures while JUnit provides a testing framework. However third party code is outside a developer’s control and thus subject to change without warning. Installing new versions of such external code/JAR files may cause developer code to break (i.e., fail to compile) in unpredictable ways. We are developing tool support to help developers adapt their code to the changes brought about by such updates. We are conducting an experiment to evaluate our approach.

What Will I Be Asked To Do
To participate in this experiment, you must be comfortable using the Eclipse integrated development environment for Java development. If you agree to participate in this study, you will be asked to perform between two and four tasks and fill out a questionnaire. One questionnaire will be given at the start of the study to ascertain your experience level and ask some general development questions. Following which, short training tasks will be used to familiarize you with the experimental tool. The main portion of the study will be two to four larger tasks. In each task you will be asked to update source code, which was written using an old version of a library, to become compliant with a new version of that same library. You will be asked to perform the task using any manual modifications you wish supplemented by either the tools already available in Eclipse or with our experimental tool. The final questionnaire will ask about your impressions of the task in addition to the overall impressions of the relative strengths and weaknesses of the experimental tool versus standard tools.

You should be willing to provide information relevant to your software development ability (e.g., the number of years of experience you have with a particular technology), and you will be asked about your opinions or experiences regarding the tasks you performed, and our tool.

Your participation in this study is voluntary, and you may refuse to participate altogether, or may withdraw from this study at any time. If you do withdraw from the study, any information you have contributed to that point may be retained.

The experiment will take less than 3 hours of your time to complete. You will be given a $20.00 honorarium when the experiment is complete.
What Type of Personal Information Will Be Collected?

Your name will be collected, but it will not be publicly reported. You will be asked to provide information relevant to your software development ability (this may include the number of years of experience that you have with certain software applications or programming languages).

Are there Risks or Benefits if I Participate?

There are no known risks to your participating in this research project.

Your identity, and any feedback you provide, will be anonymized by labelling these with generic identifiers (e.g., “Subject 1”). No further attempts at hiding your identity will be made. Note that details of your reported tasks or statements may allow some individuals (i.e., friends, co-workers) to suspect your participation.

The benefit to participating in this study is that you will be furthering research in software development tools that are designed to assist developers such as yourself. We are particularly interested in incorporating your feedback into future releases of the tool.

What Happens to the Information I Provide?

Your contributions (including your comments, responses, indications of your performance, and the output of tasks) may be publicly reported.

You will not be audio- or video-taped for this study.

Physical copies of raw data will be stored in a locked location, and raw data stored on a computer will be in password-restricted locations. This data will be accessible only to the researcher and his supervisor, and will be archived for a term not to exceed five years, for possible future use.

Signatures (written consent)

Your signature on this form indicates that you 1) understand to your satisfaction the information provided to you about your participation in this research project, and 2) agree to participate as a research subject.

In no way does this waive your legal rights nor release the investigators, sponsors, or involved institutions from their legal and professional responsibilities. You are free to withdraw from this research project at any time. You should feel free to ask for clarification or new information throughout your participation.

Participant’s Name: (please print) _____________________________________________

Participant’s Signature __________________________________________ Date: ______________

Researcher’s Name: (please print) _____________________________________________

Researcher’s Signature: ________________________________________ Date: ______________
Questions/Concerns

If you have any further questions or want clarification regarding this research and/or your participation, please contact:

Puneet Kapur  
Dept. of Computer Science  
pkapur@ucalgary.ca  
Supervisor: Dr. Robert J. Walker, Dept. of Computer Science, walker@ucalgary.ca

If you have any concerns about the way you’ve been treated as a participant, please contact Bonnie Scherrer, Ethics Resource Officer, Research Services Office, University of Calgary at (403) 220-3782; email bonnie.scherrer@ucalgary.ca.

A copy of this consent form has been given to you to keep for your records and reference. The investigator has kept a copy of the consent form.