Refactoring References for Library Migration

Puneet Kapur Brad Cossette Robert J. Walker

Department of Computer Science
University of Calgary
Calgary, AB, Canada
{pkapur, bcossett, walker}@ucalgary.ca

Abstract

Automated refactoring is a key feature of modern IDEs. Existing refactorings rely on the transformation of source code declarations, in which references may also be transformed as a side effect. However, there exist situations in which a declaration is not available for refactoring or would be inappropriate to transform, for example, in the presence of dangling references or where a set of references should be retargeted to a different declaration.

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.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; D.2.6 [*Software Engineering*]: Programming Environments

General Terms Human Factors, Languages.

Keywords Dangling references, library migration, refactoring, flexible search, flexible transformation, Trident.

1. Introduction

Automated refactoring is a key feature of modern integrated development environments (IDEs). Existing refactorings involve the transformation of source code *declarations*, such as class or method declarations, where any *references* (e.g., method calls, the use of a type name within a variable declaration) can optionally be updated to reflect changes to the underlying declaration. However, 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 the 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 application programming interface (API) has changed, dangling references will now be present in the developer's code. (2) In test-driven development, 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, 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 (a) must operate 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.

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 [26], can be excessively rigid about syntactic details [7], or can fail in the presence of broken semantics [5]. Previous work to support library migration has focused on: strong notions of type correctness [2, 20], leading to poor performance or difficulty in specifying transformations; adding adaptor layers [12, 21, 22], with runtime overhead and limited applicability; or solely on recom-

mending alternative calls [9, 27], rather than actually performing transformations. Even in the few cases where techniques attempt to actually migrate between library versions (e.g., [1, 11]), they demand access to the source code and/or a repository of example usage code, and are incapable of providing transformations in all cases—after all, inferring library migration is reducible to functional equivalence, an undecidable problem in general.

We focus here specifically on the library migration scenario. We begin by presenting an empirical study on a large number of versions of three industrially relevant software libraries, to illustrate that their APIs do indeed break between versions and characterize how.

We also present a prototype tool, called Trident, to support the refactoring of references that is intended to handle a subset of the API breakages that we have observed. 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 [8] 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.

We performed case studies with two industrial developers, who were 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.

The remainder of the paper is structured as follows. Section 2 describes an actual industrial scenario of library migration. Section 3 presents our empirical study into API breakages in a set of industrial software libraries. Section 4 describes our approach to supporting some of these transitions, as embodied in the Trident tool, which is evaluated through two in-depth case studies with industrial developers in Section 5. Section 6 discusses remaining issues and future work. Section 7 considers how our approach differs from existing work for this practical problem.

This paper contributes an empirical study into the nature of API change in a set of industrial systems, and an approach that allows references to be refactored, independent from declarations, and even when the references dangle.

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. This class library migration was not so simple in practice.

A comparison of the library versions reveals that there are 120 *binary incompatibilities* [14] 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 previously 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. The full set of changes and resulting errors in the Chartwell codebase are available elsewhere [16].

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.

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 code base that correspond to method decla-

[1] Although a beta-version, JDOM was the cutting edge technology of the day; alternative libraries were not considered viable by Chartwell.

rations 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 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 XMLOuputter(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.

3. API Change in the Wild

Various authors have mentioned the existence of API changes in practice. In particular, Dig and Johnson's study on API evolution [10] is prominent. Unfortunately, their study has three shortcomings from our perspective: (1) the number of migrations that they considered consist of a single version transition for each of 5 systems; (2) they considered only `public` entities and not all entities that a developer could depend on, and furthermore, only those entities with publicly available documentation and intended for reuse (i.e., internal packages in Eclipse would be ignored); and (3) having decided previously that library migration paths can be described via automated refactorings in the majority of cases, they do not look for changes that would contradict this notion.

Instead, we present an in-depth investigation of API changes in practice, without any presupposition about the nature of such changes. We sampled the growing body of open-source software available on the Internet, to select three systems that are heavily used in industry and generally regarded as of reasonable quality; we chose HTMLUnit, JDOM, and log4j. (As an example, HTMLUnit has been downloaded 69,343 times for the versions that we have investigated.) For each of these systems, we sought out as many versions as we could, in order to examine the API changes between successive versions.

To detect and analyze API changes, we performed 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. We concerned ourselves only with changes that could potentially break an existing client, discarding all other events. Changes considered pertinent include visibility reduction of entities (types, methods, constructors, or fields), deletion of entities, movement of entities (including formal parameters), and the insertion of entities that would break existing references (e.g., new formal parameters, new declared exception). The detailed results are tallied in the appendix.

Table 1 summarizes the results over all transitions between successive versions; detailed results are available elsewhere [16]. 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 qualitatively found that the `@deprecated` tag is an unreliable guide: deprecated entities of course do not always get eliminated, but 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.

System	Version	All		Types		Methods		Constructors		Fields	
	transitions	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
HTMLUnit	31	26.19	54.30	4.61	4.40	20.06	52.96	1.29	2.56	0.23	0.67
JDOM	8	27.88	42.80	1.25	1.28	18.50	28.75	2.75	6.98	5.38	7.10
log4j	15	13.87	39.84	1.53	3.54	7.40	23.10	0.53	1.25	4.40	12.48

Table 1. 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.

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 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.

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 [2, 6, 20, 24]. A developer is left with little choice but to manually transition their code, a notably painful process [3]. More pragmatic support would be a boon.

4. 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 allowing for intelligent replacements in contexts where existing support could not operate: incomplete code bases with dangling references.

4.1 Goals

To enable Trident to provide refactoring support for references, we aimed to address two key issues. First, we wanted to provide refactoring support equivalent in capability to existing refactoring tools, and not simply extend lexical/Java Search to include a replace feature. After spending some time observing Eclipse users invoke 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.

Context Sensitive. The types of standard refactorings available depend on the broader context in which the example resides. Selecting a variable declaration inside a method body makes available the "extract method" refactoring while selecting a similar statement in the class body does not.

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 Clause. 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 which, 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.

4.2 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, via a running example: the statement `Category.getInstance(ResourceBundleTest.class)`, in which `Category` and `ResourceBundleTest` are types.

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

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 [8] is applied. Currently only 6 types of AST node selections are supported: SimpleName, SimpleType, QualifiedName, QualifiedType, MethodInvocation, and ClassInstanceCreation.

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 name portion is `getInstance`.

Arguments have the same three search options available with one small difference. If there are ≥ 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.

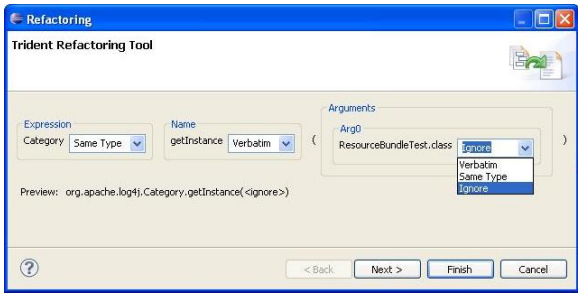

Figure 1. Trident search configuration.

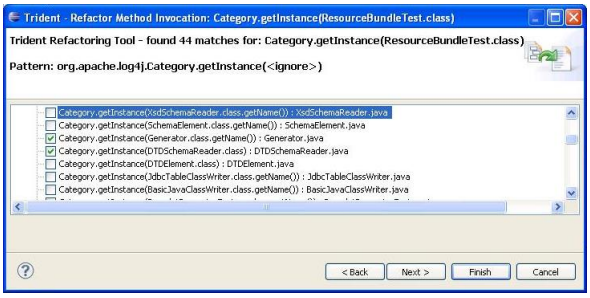

Figure 2. Trident search results.

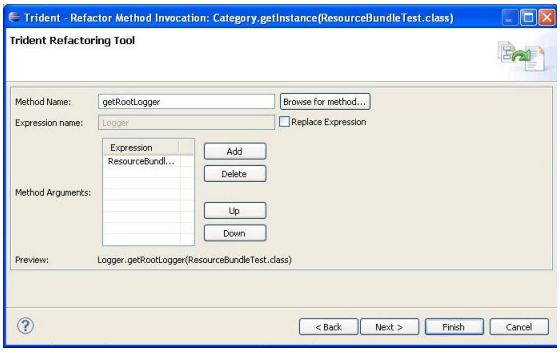

Figure 3. Trident refactoring configuration.

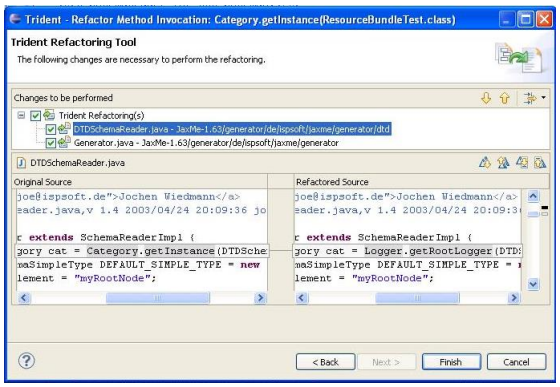

Figure 4. Trident reference refactoring preview.

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

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 2, 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.

Refactoring Configuration. The developer can then specify how the components should be altered, as illustrated in Figure 3. 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.

Preview. 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). If the previewed transformation is unacceptable at any point, the developer can unselect that location, 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 desired.

5. Case Studies

To determine whether even partial tool support for refactoring references is likely to be beneficial in practice, we conducted two case studies with 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. Section 5.2 describes our qualitative observations of the participants undertaking their refactoring tasks using Trident to assist them, and subsequently performing the same refactoring manually. Section 5.3 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 which was currently in an uncompilable state due to changes within a depended-upon library. The goal of the refactorings was to restore the system to a compilable state.

5.1.1 Participants

Participant 1 described himself as having 9 years of experience in developing Java software, and having used the Eclipse IDE for the past five years. Participant 2 described himself as having 7 years of experience developing Java software, and having used the Eclipse IDE for two years.

5.1.2 Systems

We chose to have the developers migrate the JaxMe project (version 1.63), an open source implementation of the Java Architecture for XML Binding (JAXB) specification, which defines how an XML schema can be transformed into a set of Java classes and interfaces for programmatic manipulation. The JaxMe v1.63 contains 213 classes and 18928 LOC. 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 provides application logging functionality to developers. In place of using `System.out` or `System.err` method 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 `Category` class to access the "category" of functionality that they wished to log. However, in the transition from v1.2.8 to v1.3, the functionality provided by the `Category` class was largely supplanted by the `Logger` class, because using a "logger" for logging made more sense than using a "category" (according to the manual). Similarly, the `Priority` class was originally used by developers to define the importance of a logging message being sent to the `Category` ob-

jects used for logging. In the new version, the type was replaced with a new `Level` class which allowed developers to set the *error level* of logging messages sent.

5.1.3 Task

For the case study, each developer was asked to migrate the JaxMe project from v1.2.8 to v1.3 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`.

There are several other differences between log4j v1.2.8 and v1.3, 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 v1.3 by providing stubs for missing entities that were not in the limited subset.

5.1.4 Setup

Participants were provided with the Eclipse IDE v3.5, 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.

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. Participants were then asked to modify the source code to eliminate the dangling references left by the log4j library transition, reducing the compilation error count to 0. To alleviate the developers' 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 used 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 effect, and thus give a small advantage to developer performance in the manual treatment.

For the tool treatments, participants were asked to only use 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 Observations

We focus here on a few high-level observations of the participants' actions and comments. A more complete set of observations can be located elsewhere [16].

5.2.1 Participant 1

Tool treatment. The participant's approach can be characterized as heavily iterative, cycling between selecting different exemplars as input to Trident, refining his search criteria, and investigating the search results. For a particular transformation, he discovered that providing Trident with different variations on the same input (e.g., searching on the `ERROR` field name vs. searching on the fully qualified type and field name `org.apache.log4j.Priority.ERROR`) allowed him to refine and discriminate the search results of the tool to reflect his original intention. In most cases, the participant would try several different exemplars as the input for Trident, before finding the example which captured precisely those locations he wished to transform. Later in the task, the participant exemplars in a deliberately exploratory fashion; he would select an example, activate Trident, then alter how Trident restricted its search on that example (e.g., ignoring the type or presence of a parameter in a method invocation "to use Trident to find the different kinds of arguments being passed in, before I decide on what changes to make.").

The participant used the transformation preview screens to ensure that his intended modifications were carried out correctly, but also began to use them very quickly to locate unusual cases where he wanted to see if Trident performed as they expected. For example, the participant had to perform a transformation on a static

method invocation (i.e., from `Category.getRoot()` to `Logger.getRootLogger()`), but noticed that in the JaxMe source code static methods were more often accessed on an instantiated object (i.e., `cat.getRoot()`) than through the class. The participant chose as their exemplar `cat.getRoot()` because it described the more frequently occurring situation, but used the preview screens to see that Trident (1) also captured the cases in which the method was accessed statically through the class type, and (2) correctly transformed those cases as well.

Towards the end of the task, the participant preferred to use Trident to enact large sets of changes, while using manual modification to fix "one-off" changes. In one case, he aimed to transform a method invocation which accepted a parameter whose value was almost always passed in as a static field reference on a type, but in a few cases was a reference to a local object. Rather than restrict Trident to deal with those cases separately, he reasoned instead that the side-effects of the transformation for those special cases were easily dealt with manually afterwards.

Manual treatment. After realizing that Eclipse's Java Search tool does not support search and replace operations, the participant resorted to using the File Search and Replace tool, which also allowed him to use regular expression patterns for the transformations.

The participant found he needed to iterate through several versions of a regular expression before he could get it to match examples he knew about in the source code—despite Eclipse providing inline regular expression assistance. These examples also proved to be problematic; in tailoring his patterns to these examples (which exemplified the majority of cases), he would often capture cases that were (1) sufficiently lexically similar that imprecision in the pattern caused an unintended match (e.g., forgetting to escape the dot ('.') operator because the mistake in the pattern still worked on his example), or (2) syntactically similar yet semantically different, such as patterns written to capture static field accesses on types that also captured static method invocations on the same. These consequences prompted him to comment, "The big problem with `grep` and regular expressions is that you capture things that you don't expect".

A frequent point of frustration for him was that the compilation error count in the project would barely decrease after enacting a particular transformation; the participant often found that a single transformation was not enough to resolve a single problem. Instead, he needed to apply a sequence of patterns before the transformation was complete.

For example, after a transformation was enacted, the compiler often could not recognize the new types or methods involved as their containing class had not been imported. In most cases, he was able to fix this automatically using Eclipse's Organize Imports tool. In one particular case though, this didn't work: two classes with the same name, but different enclosing packages, resided on the class path (`org.apache.log4j.Logger` and `java.util.Logger`), and the Organize Imports tool gave the error "Ambiguous references, user interaction is required." It was not able to decide which `Logger` class was being referenced by the newly transformed source code. In this particular case, the transformation had affected 44 files, requiring the manual addition of 44 import statements. To fix this, the participant chose to write a regular expression to insert the appropriate import statement after the package declaration in a Java class; because there was no way to restrict the insertion of this pattern to only those 44 files which needed it, he applied the pattern to every class in the JaxMe system, and then used Eclipse's Organize Imports to remove the unnecessary import statements from those classes in which they did not belong; he evaluated this solution as "an ugly hack." Even then, he also unintentionally transformed a string literal in the source code which happened to match his regular expression. He only caught this mistake because the transformation at that point caused a compilation error.

Regarding his experiences using regular expressions for this task, the participant noted that he "never got the [regular expressions] right the first time, despite knowing the change that had to be made," and he "encountered significant frustration from all the unintended side effects".

5.2.2 Participant 2

Tool treatment. The participant initially was confused about how to select exemplars, but this was resolved quickly by the experimenter providing a tutorial on the matter. He also failed to realize initially that multiple classes with the name `Level` existed on the classpath; this is an issue that arose from Eclipse and not specifically from Trident.

This participant can be characterized as more cautious and less willing, initially, to experiment with Trident in unexpected situations. When searches returned a broader set of hits than he had anticipated, he was tempted to restrict the locations to only those that he had anticipated and deal with the others manually. Two features of Trident led him to overcome this hesitancy: (1) global "undo" caused such unexpected changes to be low risk to try out; and (2) the introduction of comments by Trident that explained what it had changed and why helped him to understand the rationale for the broader set of results. Earlier presentation of rationale (i.e., during Restriction) would likely have helped him.

The participant asked if it was possible to restrict the search to only certain kinds of entities (i.e., type declarations within variable declaration statements, as opposed to within catch clauses, method declarations, etc.). Trident does not currently provide such a restriction capability, and he was able to complete all but 5 remaining errors with Trident. Thus, it is not clear whether this capability is needed.

Manual treatment. The participant attempted to reason about some of the API changes as though they constituted refactorings (e.g., moving fields between types). He at-

tempted to utilize the Eclipse automated refactorings, to be confronted with the error message, "Destination type does not exist", since these operate on declarations and not dangling references.

His second attempt involved Eclipse's Java Search tooling, but this did not allow him to search-and-replace. Thorough browsing of the menus led him to a lexical replacement option under Eclipse's File Search tool.

For the API change from `Category.getInstance(...)` to `Logger.getLogger(...)`, this tooling caused the participant 2 key problems: (1) the large number of hits led him to exhaustively review each before he was willing to change each; and (2) the import statements in each file had to be manually adjusted before the references stopped dangling. "I am beginning to realize what a pain this really is. Initially I had thought Eclipse search and replace would do this for me."

For the API change from `Category.getRoot()` to `Logger.getRootLogger()`, a lexical search-and-replace was again used; this was a viable option despite the fact that the `getRoot()` method is sometimes invoked on static fields, and sometimes invoked directly on the class. In all the cases involving static fields, the name of the field is consistently "cat", making lexical search-and-replace relatively straightforward. He encountered a key usability problem in the process, however: a copy-and-paste error in the replacement text field caused an extra, empty pair of parentheses to be attached to the end of the new method invocation. As the Eclipse File Search tooling does not provide a global undo capability, the participant attempted a second round of search-and-replace to fix his previous mistake.

For the API change from `static Category cat = ...` to `static Logger cat = ...`, his previous refactorings had altered the target statements to `public static Category cat = Logger.getLogger(...)`. He specified the regular expression "`Category\s+cat\s*=\s*Logger.getLogger`" to search for matches, and "`Logger cat = Logger.getRootLogger()`" as the pattern to replace it with. The replacement ought to have been "`Logger cat = Logger.getLogger()`". A series of mistaken keystrokes and button presses while attempting further regular expression-based search-and-replaces compounded the problems. "This is a nightmare. Is it okay if I quit this task?" After additional attempts, he eventually fell back to manually correcting these and all remaining errors, noting "the task seemed easy but the [manual change process] was really messy and I am not confident in the solution".

5.3 Results and Analysis

The quantitative results are given in Table 2. For both participants, Trident clearly outperformed the manual treatment, despite the study being biased in favour of the latter. Overall time for performance was reduced by 23% in using Trident; note that this time includes the significant delays incurred waiting for (the completely unoptimized) Trident tooling to complete its searches. Manual file modifications were reduced by 82–95% through the use of Trident. Interestingly, only about half of the attempted invocations of Trident were carried through to completion. This may indicate that the developers did not understand well enough the consequences of their selections before reviewing the preview results, or it may indicate difficulties with the tool interface; the qualitative observations discount the latter possibility.

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

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 effect, 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,

Participant	Time (min.)		File modifications		Trident invocations	
	manual	Trident	manual	Trident	started	completed
1	74	57	40	7	28	14
2	130	100	104	5	21	10

Table 2. Quantitative results from the case studies.

`Category cat`) to make the task of find 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 having any effect.

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 Trident.

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.

6. Discussion

6.1 Threats to Validity

Having each participant repeat the same migration task obviously calls into question the validity of the results from the second treatment, since learning effects would 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 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 such search and replaces. 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 accidentally 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 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. 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 particularly 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 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.x()` becomes `A.y()`) or API changes can be 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 initiate a refactoring is also an issue within Eclipse [19].

By default Trident scans the entire code base during the search query sub-stage which introduces a perceptible time delay in the use of the tool; Trident is not optimized in the least, at present. Providing developers with the option of limiting the number of files—by name or by package—included in the search could help alleviate this problem. Investigating the usefulness of the approach took priority over improving its performance.

6.4 Future Work

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".

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.

7. Related Work

Previous work that aims to address the problems of library migration can be classified as helping with discovering refactorings, automatic generation of adaptor code, or assisting with source code transformation.

Discovering Transformations. Some approaches provide tools to discover what kinds of transformations have been made in the library between versions, which a developer can then use to consider how to update their source code [17, 25]. Such approaches have three large drawbacks: (1) they cannot correctly handle all situations, thus requiring developer intervention; (2) they do not actually transform the devel-

oper's source code; and (3) they cannot operate in the absence of the library versions' source code.

Automatic Generation of Adaptors. Other researchers have looked into providing backwards compatibility between successive library versions by generating adaptor layers to mimic the interface and behaviour of previous versions [12, 21, 22]. Such tools are aimed at library developers, who have access to the repositories containing the history of their library's 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. Furthermore, (1) some kinds of library changes cannot be hidden behind an adaptive layer but need to be addressed in the client code; and (2) the performance overhead from the adaptors is unacceptable in some situations.

Assisting with Source Code Transformation. Source code transformation techniques 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., [26]). 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 [7] 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 semantically-aware program transformation (e.g., [13]) demand the presence of formal specifications of the source code, which tend to be absent in industrial settings. Semantic grep [5] suffers from being burdensome on the developer just as with grep.

Chow and Notkin [6] require that a library maintainer annotate changed functions with rules used to generate transformation tools. Tip et al. [24], Balaban et al. [2], and Nita and Notkin [20] require that the developer using the library write a specification of the transformation to apply in migrating from one library to another. There are 3 problems with such 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 [4] As Murphy-Hill says [18], "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."

CatchUp! [15] provides a means to record automated refactorings and replaying these so that library migration becomes automatic; this idea has been incorporated as refactoring scripts in industrial IDEs. Unfortunately, it requires that the library changes be performed with automated refactorings, which is not always possible.

Dig et al. [11] take the CatchUp! notion of recording and replaying refactorings one step further by first inferring the refactorings between two library versions, rather than demanding that the library developer record them. The key drawbacks to the technique is that not all transformations are expressible as combinations of the standard automated refactorings, and it applies heuristics to estimate refactorings in many cases, which can be incorrect. Thus, developer intervention is still necessary.

Tansey and Tilevich [23] focus on the more limited problem of refactoring annotations. Their machine learning technique requires the provision of examples, which could be a more costly technique overall than manual intervention, and it still does not provide complete coverage.

Andersen and Lawall [1] 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.

Diff-CatchUp [27] and SemDiff [9] 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 replace them: such recommenders are complementary to our work.

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. We examined many versions of a few industrially-relevant software systems and found that API changes can be frequent, without warning, and severe.

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.

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.

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.

Acknowledgments

We wish to thank Rylan Cottrell for useful feedback on earlier versions of this manuscript. This work has been supported by graduate scholarships from the Alberta Informatics Circle of Research Excellence, and by postgraduate scholarships and a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. Andersen and J. L. Lawall. Generic patch inference. *Automat. Softw. Eng.*, 17(2):119–148, 2010.

[2] I. Balaban, F. Tip, and R. Fuhrer. Refactoring support for class library migration. In *Proc. ACM SIGPLAN Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pages 265–279, 2005.

[3] M. Boshernitsan and S. L. Graham. iXj: Interactive source-to-source transformations for Java. In *Companion ACM SIGPLAN Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pages 212–213, 2004.

[4] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, pages 567–576, 2007.

[5] R. I. Bull, A. Trevors, A. J. Malton, and M. W. Godfrey. Semantic grep: Regular expressions + relational abstraction. In *Proc. Working Conf. Reverse Eng.*, pages 267–276, 2002.

[6] K. Chow and D. Notkin. Semi-automatic update of applications in response to library changes. In *Proc. IEEE Int. Conf. Softw. Maintenance*, pages 359–368, 1994.

[7] J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006.

[8] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proc. ACM SIGPLAN Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pages 313–328, 2008.

[9] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. Int. Conf. Softw. Eng.*, pages 481–490, 2008.

[10] D. Dig and R. E. Johnson. How do APIs evolve? A story of refactoring. *J. Softw. Maint. Res. Pract.*, 18(2):83–107, 2006.

[11] D. Dig, C. Comertoglu, D. Marinov, and R. E. Johnson. Automated detection of refactorings in evolving components. In *Proc. Europ. Conf. Obj.-Oriented Progr.*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428, 2006.

[12] D. Dig, S. Negara, V. Mohindra, and R. Johnson. ReBA: Refactoring-aware binary adaptation of evolving libraries. In *Proc. Int. Conf. Softw. Eng.*, pages 441–450, 2008.

[13] M. S. Feather. Reuse in the context of a transformation-based methodology. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1: Concepts and Models, chapter 14, pages 337–359. Addison-Wesley, 1989.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*, chapter 13: Binary Compatibility. Addison-Wesley, 3rd edition, 2005.

[15] J. Henkel and A. Diwan. CatchUp!: Capturing and replaying refactorings to support API evolution. In *Proc. Int. Conf. Softw. Eng.*, pages 274–283, 2005.

[16] P. Kapur, B. Cossette, and R. J. Walker. Refactoring references for library migration—Appendix. Technical Report TBD, Department of Computer Science, University of Calgary, 2010.

[17] M. Kim, D. Notkin, and D. Grossman. Automatic inference of structural changes for matching across program versions. In *Proc. Int. Conf. Softw. Eng.*, pages 333–343, 2007.

[18] E. Murphy-Hill. A model of refactoring tool use. In *Proc. Wkshp. Refactoring Tools*, 2009.

[19] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for Extract Method. In *Proc. Int. Conf. Softw. Eng.*, pages 421–430, 2008.

[20] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proc. Int. Conf. Softw. Eng.*, 2010. In press.

[21] I. Şavga and M. Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proc. Int. Conf. Generative Progr. Component Eng.*, pages 175–184, 2007.

[22] I. Savga, M. Rudolf, S. Goetz, and U. Aßmann. Practical refactoring-based framework upgrade. In *Proc. Int. Conf. Generative Progr. Component Eng.*, pages 171–180, 2008.

[23] W. Tansey and E. Tilevich. Annotation refactoring: Inferring upgrade transformations for legacy applications. In *Proc. ACM SIGPLAN Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pages 295–312, 2008.

[24] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for generalization using type constraints. In *Proc. ACM SIGPLAN Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pages 13–26, 2003.

[25] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. IEEE/ACM Int. Conf. Automat. Softw. Eng.*, pages 231–240, 2006.

[26] S. Wu and U. Manber. Agrep—A fast approximate pattern-matching tool. In *Proc. USENIX Winter Technical Conf.*, pages 153–162, 1992.

[27] Z. Xing and E. Stroulia. API-evolution support with Diff-CatchUp. *IEEE Trans. Softw. Eng.*, 33(12):818–836, 2007.