Seeking the Ground Truth: A Retroactive Study on the Evolution and Migration of Software Libraries

Bradley E. Cossette and Robert J. Walker
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada
{bcossett, walker}@ucalgary.ca

ABSTRACT

Application programming interfaces (APIs) are a common and industrially-relevant means for third-party software developers to reuse external functionality. Several techniques have been proposed to help migrate client code between library versions with incompatible APIs, but it is not clear how well these perform in an absolute sense. We present a retroactive study into the presence and nature of API incompatibilities between several versions of a set of Java-based software libraries; for each, we perform a detailed, manual analysis to determine what the correct adaptations are to migrate from the older to the newer version. In addition, we investigate whether any of a set of adaptation recommender techniques is capable of identifying the correct adaptations for library migration. We find that a given API incompatibility can typically be addressed by only one or two recommender techniques, but sometimes none serve. Furthermore, those techniques give correct recommendations, on average, in only about 20% of cases.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement.

General Terms

Experimentation, design, measurement, verification.

Keywords

API, adaptive change, recommendation systems.

1. INTRODUCTION

Software developers rely on external libraries to provide reusable functionality. Ideally, developers can rely solely on the application programming interface (API) that libraries provide to access their functionality, without worrying about their implementation details [19]; they hope that the API is a contract that will not change [3, 17, 12, 2]. However, software libraries are susceptible to the same environmental pressures to change that all software systems face [18, 21, 22]. As a result, library maintainers need to evolve their systems in ways that sometimes result in an incompatibility between the old and new versions of their API [12, 22]. This raises a dilemma for both API developers and client developers: whether to migrate to the new API version and endure the adaptive effort, or to refuse to migrate to the new version—risking the problems, bugs, and security risks that come from relying on obsolete software.

Much research has addressed the issues involved in migrating source code to accommodate API changes [3, 10, 12, 6, 27, 16, 8, 20, 26, 15, 5]. Yet only a few papers empirically study how APIs change over time, classifying the nature of the changes these systems undergo [7, 15]. Despite numerous evaluations of change recommendation or library migration approaches, there exists no established corpus of unbiased library change data. To both fully appreciate the complexities of real library migration issues, and to serve as a standardized basis for evaluating novel solutions, we need a collection of all points of breaking change between a set of API versions, that also identifies the correct replacement to migrate dependent code in each case. Such a corpus would permit absolute measurements of the quality of recommendations, including knowledge of misses—important goals for this growing research area.

We present an empirical investigation into the changes of Java software libraries and their APIs across multiple revisions. We identify all points of change in each new version of an API that resulted in a *binary incompatibility* with respect to its previous version. A binary incompatibility (BI) is defined [11, ch. 13] as any change to a type such that code compiled and linked without error to a binary containing that type prior to the change, does not link to a new binary containing the changed type; lists of BI kinds are available on-line.[1] For each BI, (1) we determine whether replacement functionality exists in the new library version; (2) we manually apply six library migration techniques to each BI, to observe the ability of each to recommend correct replacements; and (3) we analyze each of the replacements we found, to classify the nature of the transformation that would need to be applied to client source code to

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA

[1] e.g., http://wiki.eclipse.org/Evolving_Java-based_APIs_2

adapt to that particular library change. We found that existing change recommendation techniques were successful, on average, in only 20% of the cases examined. Furthermore, we discovered that the majority of library changes could not have their needed transformations automatically enacted upon client source code without significant developer intervention and/or specification.

The rest of this paper is structured as follows. In Section 2 we discuss previous work on library migration. In Section 3, we present the methodology we followed in our study. In Section 4 we present our observations on the data we have collected, including quantitative descriptions of BIs, technique effectiveness, and classifications. We present a preliminary extension to an additional technique in Section 5. In Section 6 we discuss our findings, their implications, and threats to the validity of our work.

The contributions of this paper are: (1) the classification and analysis of source code transformations necessary to adapt to library change; (2) a corpus of complete library change data, for use in evaluating and replicating library migration techniques; and (3) a tool-independent comparison of change recommendation techniques.

2. RELATED WORK

Existing research on addressing the issues caused by API evolution has focused primarily on how to preserve backwards compatibility through the provision of adaptive layers (e.g., wrappers) that simulate the original, legacy interface [8, 20]. Alternatively, some work has considered how to migrate client systems to the new library version by altering the client's implementation.

2.1 Supporting Library Migration

Some work has examined how the original library developer can specify or capture the changes enacted between library versions in order to mechanically enact them on a client's source code without developer intervention. In the simplest cases, the library maintainer uses tooling to record refactorings enacted on their source code, and from this, generates a refactoring-script that can then be shipped with their new library version [12, 7]; another technique requires the library developer to provide annotated rules for changed functionality that describe how to update dependent code [3, 1]. While such techniques spare client developers a substantial amount of work, they have significant limitations as to the kinds of API evolution, and hence the corresponding migrations, that they can support. The migration approaches devised by such techniques are also done in isolation from client source code, and may not be aware of the particular characteristics of how the client uses their libraries. Other related approaches include attempts to inline missing functionality from the old library into the developer's source code, a technique that is not always appropriate [22].

Several techniques recommend how to replace broken functionality caused by an upgraded API. These techniques use one or more analysis techniques to compare two library versions, and attempt to reconcile how the versions have changed. The analyses include: the lexical comparison of method signatures [16, 26]; whether new functionality in a library is a code clone of previous functionality [25]; how a model of the underlying software is altered between versions [27]; how a library's use of its own API has changed [5]; how other developers have migrated their code or test suites to accommodate the API change [24]; and using a combination of some of these techniques [26].

2.2 Empirical Studies of API Change

There are few empirical studies on how APIs evolve. Finding actual points of change in libraries is painstaking work [4], and determining how undocumented changes to an API should be addressed requires substantial investigation. Nearly all evaluations of API migration tools utilize no *a priori* knowledge of what the actual changes and replacements should be in the libraries analyzed.

Dig and Johnson's case studies on API evolution [7] are among the few exceptions. For two versions of five APIs, they determined how many breaking and non-breaking changes occurred in each, classified each change as refactorable or non-refactorable, and provided a breakdown as to the kinds of refactorable and non-refactorable changes they encountered. An often cited statistic from their work is that "...over 80% of these changes are refactorings." [7, pp. 83] Unfortunately, their work has three main shortcomings from our perspective: (a) they analyze only a single transition between library versions; (b) they analyze only public entities—thus ignoring API specialization points provided through protected entities and methods, common in object-oriented frameworks [13]; and (c) they consider only those entities that they deem intended to be reused, and for which there was publicly available documentation—thus not supporting developers who utilize APIs in unexpected ways.

In our previous work, we report data on BIs across the evolution of three libraries, encompassing 54 versions in total [15]. We showed that libraries frequently and seriously change over time, and that even in later, supposedly mature versions, substantial change can occur. Our results only presented a coarse-grained summary of the kinds of BIs introduced in each version, and of the distribution of changes over the life of each library; we did not classify or resolve any of the changes we found.

The only other result in this area is an "estimate" by Henkel and Diwan [12] that 59% of the deprecated methods and class constructors in the Java 1.4.2 SDK could be replaced with refactorings. They provide no details of how they arrived at this figure.

2.2.1 *Experiment-based studies*

In order to evaluate API migration techniques, researchers need data about how APIs have evolved in reality so as to interpret whether the techniques' output is correct; however, to-date there is no corpus of software library change data that is publicly available for tool verification and testing, and there is only one study in which the researchers identified change set data independently from the evaluation of their API migration technique [6]. In examining the literature, common practices for approximating change set data when evaluating API migration and change recommendation tools include: searching for keywords like "refactoring" in the commit logs of version control systems [25]; restricting the evaluation to a few, well documented changes and their replacements [1, 22]; using a recommendation tool to generate predictions, then checking each prediction to see if it appears to be correct [16, 27, 24, 26]; comparing tool predictions and performance relative to prior techniques, and investigating cases where predictions differ [16, 24, 26]; and choosing a sample of negative recommendations, sometimes

representing as little as 0.55% of the entire dataset, to evaluate the recall accuracy of the technique [25]. Some techniques present no evaluation of their effectiveness [3, 12].

Other researchers have chosen to evaluate their tools and techniques through case studies in which they enacted a library migration for a few software systems [5, 15], or simulated a library migration for a subset of type changes across several software systems [1]. In these cases, performing actual migration tasks provides a fair evaluation of whether the tool and technique deals with all the necessary cases, as the compiler and/or the run-time behaviour of the program will catch missed ones; the drawback is that the changes these tools are evaluated on tend to be a small subset of the potential changes that exist in that library, or in software APIs in general. There is no reason, without digging deeper, to presuppose that these are representative scenarios.

3. METHODOLOGY

Our goal in this work is to create a corpus of API change data which describes what has changed and what it has been replaced with in the new version, of sufficient completeness and accuracy to be useful to researchers to evaluate the effectiveness and limitations of API migration approaches.

We were also interested in understanding how changes in libraries potentially impacted the effectiveness of techniques that are intended to detect such changes, or to adapt source code to them. We augmented our basic data collection with two additional analyses for each change: How effective are recommendation techniques in finding the appropriate replacement for each change? How feasible is it to mechanically transform client source code to automatically migrate each existing dependency to its replacement?

We chose to analyze five Java-based APIs: Apache Struts (versions 1.0.2, 1.1.0, 1.2.4, 1.2.6, 1.2.7, 1.2.8, 1.2.9), log4j (versions 1.0.4, 1.1.3, 1.2.1, 1.2.2, 1.2.5, 1.2.6, 1.2.7, 1.2.8, 1.2.9), jDOM (versions 1.0.b6, 1.0.b7, 1.0.b8, 1.0.b9), DBCP (versions 1.0.0, 1.1.0, 1.2.0, 1.2.1, 1.2.2, 1.3.0, 1.4.0), and SLF4J (versions 1.0.1, 1.5.1, 1.5.5, 1.5.8). We chose the Struts, log4j, and jDOM libraries for the sake of replicating previous library migration studies (of Dig and Johnson [7], of Schäfer et al. [24], and of Wu et al. [26]; of Dig and Johnson [7] and of Kapur et al. [15]; and of Kapur et al. [15], respectively); our choice of the DBCP and SLF4J libraries was influenced by their popularity, as reported elsewhere [14].

3.1 Determining Points of Change

To determine the changes within each library, we examine the binary incompatibilities introduced between successive versions of the same library. Our use of BIs offers several advantages over what has previously been attempted: the determination of what is and is not a BI can be mechanically determined from an analysis of the JAR files of two library versions, with a negligible risk of missing a breaking change between the versions; it does not rely on the correctness and completeness of documentation [7, 25]; and finally, the introduction of a BI into a new library version represents a point of change that *must* be addressed in client code to migrate to the new library. Binary incompatibility thus represents, in the general case, a necessary but not sufficient condition to overcome in migrating between libraries.

To determine the BIs between two library versions, we built an analysis tool based on Clirr, an open-source tool for analyzing precompiled API JAR files for binary compatibility.[2] For each adjacent pair of library versions, we used our tool to determine the entities (classes, methods, and fields) in the old API which were binary incompatible with the new API, thereby generating a report listing Clirr's classification of each. The final report for each API version transition was written to a file in comma-separated-value format, and imported into Microsoft Excel.

3.2 Finding Replacement Functionality

For each entity we found in the old API manifesting a BI with the new API, we examined the new API to determine if it contained an appropriate replacement. To assist our investigation, we systematically applied analysis techniques that have been proposed in the various change recommendation techniques described in Section 2.1. Each technique was applied independently of the others to evaluate its ability, in isolation, to find the correct solution for each BI.

Once all techniques were applied, if we had found a suitable replacement for the binary-incompatible entity, we recorded it as the replacement in our corpus. In cases where we found an answer, but had doubts as to whether it was correct, we marked the replacement as "unsure". In a few situations, we found multiple potential replacements. We decided that, in all such cases, we would record as the replacement the entity for which adaption of the client was easiest. Finally, for those cases in which we failed to find a replacement, we recorded the entity as "deleted".

We chose to apply seven change recommendation techniques manually, using only tooling available to us in the Eclipse IDE (v3.7). We describe these below.

Original code comments. We examined the code comment associated with the binary-incompatible entity to see if it had a deprecated tag, and/or if the comment indicated what the replacement functionality should be. We then investigated the comment's recommendation to see if it was an appropriate replacement: this step was necessary as we found a number of cases where deprecated tags referred to entities that did not exist in the new API.

Release documents. We examined the release notes (if available), and any other documentation we could find online to see if the BI had been documented, and if its replacement was specified. Similar approaches are used in the literature [7, 25] for collecting change sets.

Implementation details and dataflow analysis. We applied a series of investigative techniques on the code to determine if it was possible to reason about what the replacement for the code should be. Primarily, this entailed three things: (i) studying the implementation of the entity, as well as its usage, to reason about the intent behind the functionality; (ii) performing static dependency analysis on the entity, and any other types, methods, or fields referenced by it, including constants; and (iii) examining the inheritance hierarchy and subclasses of the containing class. This approach took considerable time and effort to apply.

Call analysis. The call analysis technique examines references to the binary-incompatible entity in the old API version, and looks to see if those references also exist in the new API: if they exist, the entity or entities to which those references now refer to in the new API may be the replacement for the BI [5]. For this technique, (i) we used

[2]http://clirr.sourceforge.net; v0.7 was used.

Eclipse's static analysis tooling to find references to each binary-incompatible entity in the older version of the library; (ii) we checked if those references also existed in the new version of the library; and (iii) if so, we examined the entity that these references pointed to in the new API to see if it would be an appropriate change recommendation. If none of the references existed in the new library version, (iv) we searched outwards via transitive references to the binary-incompatible entity, until we had found a relevant recommendation or no more transitive references existed. Similar approaches are reported in the literature [5, 26]; however, unlike these, we did not restrict our analysis to methods.

Lexical search via the code comment. We selected on the first line of text in the entity's code comment (if any such existed), and performed a text-based search in Eclipse against all Java files in the new library. We examined all the code comment matches to determine if they were also associated with an entity that was the correct replacement.

Lexical search via the entity name. We selected a portion of the entity's signature, and searched for lexical matches in the new API using Eclipse's text-based search tool. The portion selected depended on the entity in question: for classes, we selected the class name, and the `class` reserved word; for methods, the method name and its associated return type; and, for fields, the field name and its associated type. We found it necessary to add the `class` reserved word as searching without it often lead to an overwhelming number of mostly false hits.

Experience. For some cases, we were initially not able to find a replacement for the binary-incompatible entity using any of the previous six analysis techniques; however, in the course of later examinations of that API version, or subsequent version of the same, we stumbled across functionality or code comments that seemed relevant to those prior unsolved cases, and eventually led us to an appropriate replacement. These cases illustrated to us how difficult it was to "penetrate" the API [23], and as such we could only determine replacements by developing some degree of experience with the library and its change patterns.

3.3 Classifying Transformations

After each replacement was identified, we examined the differences between the binary-incompatible entity and the replacement functionality we had found, and attempted to classify the change that occurred. Initially, we used basic refactoring terminology (e.g., move, rename) to describe changes where appropriate, but as we progressed, this proved inadequate. In some cases, we determined that the change itself was a combination of other refactorings, and recorded all those that we felt applied. Finally, we considered the effect of migrating client source code to accommodate this library change, and recorded whether the resulting transformation would replace one entity reference in the original client code with one other (1:1), one reference with multiple references (1:n), multiple references with one (n:1), or multiple references with multiple, other references (n:m). In cases where the entity appeared to be deleted, we recorded it as a 1:0 transformation.

3.4 Verification of Recommendations

The data we have collected represents our best judgment of how to repair BIs introduced between each API version. With the exception of the tooling we used to detect the BIs, all of our work has been done either manually, or with the assistance of the standard tools available through Eclipse. While we have been as careful and thorough with our results as we could, there is a reasonable expectation that we have made some mistakes. To check our work, we undertook a small verification case study in which we selected a subset of the changes we identified to be reviewed by other developers. The analysis of the Struts 1.0.2 to 1.1.0 migration was the most difficult we encountered, so we chose it for verification.

We randomly selected 20% of the BIs in this Struts version transition (44 changes in total); we then selected 15% of those changes to be seeded as incorrect recommendations (7 changes total) to ensure our participants did not simply agree with our results, but took the time to investigate each. For the incorrect changes, we used the results from techniques that we had evaluated as making erroneous recommendations. One case was the `ActionMapping.setActionClass(String)` method, whose deprecation comment indicated a replacement which did not exist in the new API version. Another was the `ActionServlet.validate` field which was used to check if an older configuration format was in use, and was deleted to force a format migration in the new API; this also happened to be an exact lexical match to the `ActionConfig.validate` field, which is used to determine if a validation method on an associated object should be called or not. The remaining 5 incorrect changes were randomly selected from the review set; using the recommendation techniques described in Section 3.2, we searched for reasonable but wrong replacements that participants would likely encounter.

We recruited two PhD students with industrial and relevant web-development experience to undertake our study. Each participant was provided with both versions of the library source code, and a spreadsheet listing the BIs selected for this verification study and the proposed replacement for each, which included the incorrect changes that we had seeded. Each participant was asked to determine whether the replacement for each BI in the new API was correct or not, and to record their decision in the supplied spreadsheet; if the recommended replacement was wrong, we asked them to to indicate what they believed the correct replacement was. If the participants were unsure whether a replacement was correct or incorrect, they were asked to record "unsure". Participants were allowed to use whatever tooling they wished to assist in their work; both used the standard tools available in the Eclipse IDE. The participants each took approx. 4 hours to review the supplied 44 changes.

Participant 1 (P1) agreed with 33 of our 37 correct changes (89.1%), and was unsure of 1; however, P1 also discovered one mistake we had made, finding a better replacement for a method than we had reported. Participant 2 (P2) agreed with 35 of our correct changes (94.6%), and was unsure of 2. One of the changes that P2 agreed with was the one that P1 had determined to be incorrect. For the incorrect changes that we had seeded, P1 caught 4 of them, and missed 3, while P2 caught 3, was unsure of 2, and missed 2. The three incorrect cases that P1 missed included both changes seeded from the erroneous recommendations, and another change where they felt another replacement was correct; had they checked the scope for their choice they would have noticed it was inaccessible to the original entity and its subclasses. Participant P2 also missed both changes that were seeded from erroneous recommendations.

4. OBSERVATIONS AND RESULTS

We begin with an overview of the changes we observed in Section 4.1, followed by a discussion on the effectiveness of the change recommendation techniques we applied in Section 4.2, and ending with a description of characteristics and nature of the changes in Section 4.3.

4.1 Overview of Changes

Table 1 presents an overview of the BIs we detected across all versions of the libraries we examined, with breakdowns based on the entity type (classes, methods, or fields), and visibility (public or protected).

		Class		Method		Field	
		+	#	+	#	+	#
Struts	1.0.2–1.1.0	76	0	53	28	1	73
	1.1.0–1.2.4	35	0	114	17	0	13
	1.2.4–1.2.6	0	0	0	2	0	14
	1.2.6–1.2.7	0	0	17	0	0	0
	1.2.7–1.2.8	0	0	0	0	0	0
	1.2.8–1.2.9	0	0	0	0	0	0
log4j	1.0.4–1.1.3	6	0	9	5	11	9
	1.1.3–1.2.1	11	0	87	8	1	6
	1.2.1–1.2.2	0	0	1	0	0	0
	1.2.4–1.2.5	2	0	1	0	0	0
	1.2.5–1.2.6	0	0	1	0	1	0
	1.2.7–1.2.8	0	0	0	0	0	0
	1.2.8–1.2.9	0	0	0	1	0	0
jDOM	1.0b6–1.0b7	2	0	8	5	0	7
	1.0b7–1.0b8	3	0	27	2	0	14
	1.0b8–1.0b9	1	0	16	4	0	5
totals		136	0	334	72	14	141

Table 1: Summary of BIs. + = public entities; # = protected entities.

		0	1	2	3	4	5	6
Struts	1.0.2–1.1.0	31	130	51	15	3	1	0
	1.1.0–1.2.4	5	81	72	17	4	0	0
	1.2.4–1.2.6	0	14	2	0	0	0	0
	1.2.6–1.2.7	0	0	0	17	0	0	0
	1.2.7–1.2.8	0	0	0	0	0	0	0
	1.2.8–1.2.9	0	0	0	0	0	0	0
log4j	1.0.4–1.1.3	4	14	14	8	0	0	0
	1.1.3–1.2.1	5	8	82	18	0	0	0
	1.2.1–1.2.2	0	0	1	0	0	0	0
	1.2.4–1.2.5	3	0	0	0	0	0	0
	1.2.5–1.2.6	0	0	2	0	0	0	0
	1.2.7–1.2.8	0	0	0	0	0	0	0
	1.2.8–1.2.9	0	0	1	0	0	0	0
jDOM	1.0b6–1.0b7	2	9	8	3	0	0	0
	1.0b7–1.0b8	4	12	29	1	0	0	0
	1.0b8–1.0b9	4	5	15	2	0	0	0
totals		58	273	277	81	7	1	0

Table 2: BIs resolved by a given number of techniques.

We detected *no* BIs between versions of the DBCP and SLF4J APIs; this initially surprised us, but we found that in the case of SLF4J its maintainers had made a special effort to preserve binary compatibility across all versions of their library.[3] Consequently, we have omitted those versions from the tables.

The complete dataset represents 697 binary incompatible changes across 16 API version transitions. Incompatibilities in methods accounted for the majority (58.2%) of all API changes, with class (19.5%) and field (22.2%) incompatibilities accounting for the rest. Most of the changes were made to public entities (69.4%), but nearly a third of all BIs were caused by protected entities (30.5%), two-thirds of which were fields.

4.2 Recommendation Techniques

Table 2 presents an overview of how many of the analysis and recommendation techniques described in Section 3.2 (not including Experience) were successful in finding the right replacement functionality for each BI. Most BIs had only one (33.7%) or two (34.2%) analysis techniques that were successful in finding replacement functionality. Surprisingly, 21.2% of all BIs could not be resolved with *any* of the analysis techniques used; in these cases, we either needed to rely on our experience with the library to find the replacement, or we simply could not determine what happened to that functionality in the new library version. This may be an indication of how penetrable the APIs were [23].

Table 3 provides an individual breakdown of how often each of the analysis techniques described in Section 3.2 was successful in finding replacement functionality, how often it failed, and how often it provided us with an erroneous recommendation that had a high degree of confidence. Most techniques proved to be effective for 20% of the changes we examined, though there was not a high degree of overlap, as indicated by Table 2. We have also reported how many cases were resolved due to experience here. The remainder of this subsection discusses how each technique fared.

Call analysis. The effectiveness of call analysis was highly variable, ranging from success rates of 9% in Struts, to around 40% for log4j and jDOM. Researchers have noted this technique relies heavily on APIs referencing their own functionality to glean change examples from [5]; it would useful if measures of API self-referencing were devised that could be correlated with call analysis performance. We did notice that call analysis was the least error-prone technique, aside from release documentation. Cases where call analysis led to errors could often be described as "code hypocrisy", where the public API was altered, but the original functionality was moved and kept for internal use. For example, tracing how references to the `ActionServlet.application` field in Struts 1.0.2 were replaced in 1.1.0 leads to a recommendation that its replacement be an attribute stored in `HttpServlet` superclass, when it should be the `Action.getResources(HttpServletRequest)` method that is aware of the current module context, and returns a more appropriate attribute.

We saw several examples where call analysis failed to find a change recommendation as a caller at a later point in the call chain was also altered in the new API, such that it appeared to have been removed, for example, as with the

[3] www.slf4j.org/faq.html

		original CC			release docs.			impl. details			call analysis			lexical search via: entity name			CC	Ex
		F	M	E	F	M	E	F	M	E	F	M	E	F	M	E		
Struts	1.0.2–1.1.0	15	211	5	30	201	0	120	111	0	24	206	1	59	158	14	9	37
	1.1.0–1.2.4	59	119	1	35	144	0	122	57	0	14	163	2	47	119	13	11	4
	1.2.4–1.2.6	0	16	0	0	16	0	16	0	0	2	14	0	0	16	0	0	0
	1.2.6–1.2.7	0	17	0	0	17	0	17	0	0	0	17	0	17	0	0	17	0
	1.2.7–1.2.8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.2.8–1.2.9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
log4j	1.0.4–1.1.3	2	38	0	0	40	0	33	7	0	18	22	0	13	24	3	0	0
	1.1.3–1.2.1	48	53	12	61	52	0	64	49	0	37	76	0	14	97	2	2	0
	1.2.1–1.2.2	0	1	0	0	1	0	1	0	0	0	1	0	1	0	0	0	0
	1.2.4–1.2.5	0	3	0	0	3	0	0	3	0	0	3	0	0	3	0	0	0
	1.2.5–1.2.6	0	2	0	1	1	0	1	1	0	0	2	0	0	2	0	1	1
	1.2.7–1.2.8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1.2.8–1.2.9	0	1	0	0	1	0	1	0	0	1	0	0	0	1	0	0	0
jDOM	1.0b6–1.0b7	2	20	0	1	21	0	18	4	0	10	12	0	3	19	0	0	0
	1.0b7–1.0b8	21	24	1	0	46	0	36	10	0	16	29	1	0	46	0	0	0
	1.0b8–1.0b9	4	22	0	0	26	0	20	6	0	15	11	0	2	24	0	0	0
totals		151	527	19	128	569	0	449	248	0	137	556	4	156	509	32	40	42

Table 3: Effectiveness of techniques to detect resolutions. F = found; M = missing; E = error; CC = code comment; Ex = experience.

`ActionServlet.destroyActions()` method in Struts 1.0.2. Iterative approaches to call chain analysis, where call chains were reanalyzed as other BIs were resolved, could improve the accuracy of this technique [7]; we note that recent work has attempted this [26].

Code comment. BIs were rarely documented as deprecated, and only 21.7% of affected entities had replacement functionality described in their associated code comment. However, code comments proved to have the highest error rate of the analysis techniques we examined, with 10–20% of the documented replacements incorrect: the vast majority of these cases occurred when replacement functionality was also removed in the new API, for example, as with the `BasicConfigurator.disable(String)` method in log4j 1.1.3, whose code comment recommended `Category.getDefaultHierarchy().disable()` as a replacement, but the `Category.getDefaultHierarchy()` method had disappeared in 1.2.1. Tool support to verify the accuracy of referenced entities in code comments would help API maintainers fix oversights like this, and would likely be straightforward to create.

Release documentation. The coverage of BIs between library versions varied heavily; we had a difficult time in finding release documentation for the relevant jDOM versions, and consequently our reported effectiveness here may be artificially low. Still, the majority of binary incompatible changes between API versions were not documented, with as few as 14.8% of all breaking changes documented in Struts. In contrast to code comments, we did not find any cases where release documentation was wrong about the replacement for a breaking change, and as such we only report the successes for this technique.

Lexical search via the entity name. Across all libraries, lexical searches on entity names found appropriate replacements for 22.4% of all BIs; however, performance for individual libraries varied significantly, as only 5.3% of breaking changes in jDOM had replacements locatable through lexical searches, as compared to 27.8% in Struts. This discrepancy is likely due to two factors: the degree of uniqueness in entity naming across API versions, and the percentage of changes in APIs that did not involve renaming or replacing functionality. We also found, unsurprisingly, that in cases where lexical searches return a single result with a high degree of similarity in the match, it was wrong 17.0% of the time; an example of this is presented in Section 3.4. In these cases, lexical search is easily confused when methods, fields, or classes with similar names represent very different functionality and behaviour. More sophisticated lexical search techniques for method matching exist [16].

Lexical search via the code comment. While our approach to searching on code comments is simple, there were several instances where this was the only technique that led to us to a replacement for a breaking change, such as for the `ResetTag.getName()` method in Struts 1.0.2, which was replaced with `ResetTag.getProperty()` in Struts 1.1.0.

Implementation details. The high success rate of this technique comes at a heavy price: examining and reasoning about implementation details to determine replacements for breaking changes was the most effort-intensive analysis we undertook. We think it is significant to note that for 35.5% of all BIs, we were unable to successfully determine what the replacement should be through code analysis.

Experience. While there were few changes in most API transitions that relied on our experience with the API to resolve, the Struts 1.0.2 to 1.1.0 migration was particularly problematic to deal with. A significant problem here was the poor quality of source code documentation: only 8.7% of the breaking changes had a code comment which indicated a

replacement, and 25% of those code comments were wrong. We found that in subsequent releases of the Struts API, the overall quality of documentation improved with a greater number of BIs documented in the release notes, or in the entity's code comment, and thus it was far less necessary for us to rely on experience to resolve breaking changes.

4.3 API Transformations

In classifying how each API changed between versions, we borrow terminology from the refactoring community to describe cases that clearly resembled meaning-preserving restructuring of source code [9]; in others, we attempt to succinctly describe the transformation that occurred, and we cluster related changes under a more generalized term when appropriate. In cases where the transformations appeared to be a composite of other transformations, we applied multiple classifications to the change: 170 of the BIs we examined were composite transformations, and consequently we identified 1042 transformations that occurred across 697 BIs.

We have grouped these transformations according to the degree to which we believe that they can automatically enacted on client source code, as opposed to requiring some degree of specification or input from a developer on how to resolve semantic problems: fully automatable transformations (Section 4.3.1), partially automatable (Section 4.3.2), and hard to automate (Section 4.3.3).

4.3.1 Fully Automatable

Fully automatable transformations involve API changes that do not alter the semantics of the API significantly, and thus can be sufficiently reasoned about or captured by an API migration tool such that additional developer input is not needed to migrate dependencies on the binary incompatible entity to its replacement(s). We classified Move, Rename, Pull Up, Remove Parameter, Parameters Reordered, and Encapsulate transformations as being fully automatable. Descriptions of these transformations are readily available (e.g., [9]), and we do not elaborate on them here, except to note that we did not restrict Encapsulate transformations to fields. In the log4j's `JMSSink` class, a substantial portion of the functionality was moved out of the main method body and into the class's constructor between 1.2.5 and 1.2.6, simplifying a series of configuration steps.

4.3.2 Partially Automatable

Partially automatable transformations involve API changes that, in the general case, would be difficult to enact without additional specifications and input from a developer; however, in particular cases, entities have unique structural characteristics, or assumptions about their mode of use, that could allow migrations to be automatically enacted provided a series of preconditions are met.

Shorten Subtype Chain involves cases where the leaf subtype of an inheritance hierarchy was replaced with its immediate supertype. Mechanically enacting this change would normally require developers to determine how to remap methods, fields, and alternative behaviours that may have been present in the subtype that do not exist in its supertype; however, in cases like the `ActionError` class in Struts 1.1.0, some subtypes consisted entirely of constructor calls which were delegated to its supertype. While the general case cannot be automatically transformed, these cases could.

Encapsulate (Partial) refers to changes in which functionality was encapsulated, but its replacement did not provide the same set of functionality as previously existed. For example, the `LoggingEvent.locationInfo` field in log4j 1.0.4 was changed from a public field to private in 1.1.3, and a public getter method was provided, but not a setter. These transformations seem to occur when either (a) the API maintainers need to prevent access to functionality they should never have exposed in the first place, or (b) the maintainers feel that a portion of the functionality is likely not being used (e.g., they assume client source code is not altering the contents of the `locationInfo` field). Enacting this change on client code will not automatically work if the callers are using the encapsulated entity in ways the API maintainers need to prevent, or did not anticipate: in these cases, client developers will need to rework their code, perhaps significantly, to accommodate this change.

Expose Implementation involves several related transformations, which generally stemmed from a convenience method or class being removed or broken up, but the underlying functionality was still accessible to client code: client callers were now expected to implement the exposed functionality directly, rather than rely on the library to provide convenience methods. For instance, in jDOM 1.0b7, the `Document` class provided several convenience methods, such as `getProcessingInstruction(String target)` which searched a content list for the first `ProcessingInstruction` found, and returned that to the caller. In the 1.0b8 API, callers were instructed to instead directly access the content list via `getContent()`, and perform the search themselves. In cases like this, client code can by migrated by simply inlining the API's implementation of the method (e.g., [22]).

In some cases though, the exposed implementation made several assumptions about the objects and data that the client code has access to in its calling context, which may not be true if the caller was relying on the API to gain access to them. For example, subclasses of `FormBeanConfig` in Struts 1.2.4 would need access to an `HttpServletRequest` object to obtain the data that in v1.1.0 was stored in the `moduleConfig` field. In other cases, the exposed implementation took the form of a new abstract class (e.g., `ConditionalTagBase` in Struts 1.1.0) which made it easier for the developer to provide their own specialized implementation of the missing functionality (e.g., `IfAttributeExistsTag`), but would require some insight to recognize the one or two key lines of code that need to be added to a new subclass to implement the new solution.

Consolidate consists of transformations in which several classes, methods, or fields were replaced by more generalized functionality; it is essentially the inverse of Expose Implementation. Problems arise though when the consolidated functionality includes additional operations or behaviours that may not have been expected by client callers, such as with the consolidation of the `ActionServlet.initDebug()` method in Struts 1.0.2 into the `ActionServlet.initOther()` method in 1.1.0. Some degree of restriction would be needed to ensure that replaced client code does not have new behaviours introduced which the client code did not expect.

4.3.3 Hard to Automate

Hard-to-automate transformations are cases that represent significant alterations to the semantics of entities between

versions; a tool is unlikely to handle such a transformation without assistance from a developer (e.g., [3, 1, 20]).

Replace by External API involves changes in which the library removed functionality it previously provided, and expected client callers to replace that functionality with equivalent types from other libraries. This introduces two problems: first, the client developer must determine how to incorporate the external API if it does not ship with the new API version, which happens in cases where the API adopts a facade to generically interact a variety of libraries (e.g., Struts adoption of the Apache Commons Logging facade to replace its internal logging implementation); secondly, the replacement is often designed with generic reuse, and may be missing functionality that was specific to the previous API, or may require initialization steps that were previously taken care of by the API.

Add Contextual Data involves changes, typically to methods and to class constructors, in which additional information in the form of a new parameter was required in the new API. Unlike the "Add Parameter" refactoring [9], these changes clearly expect that callers can gain access to the required data, which is non-trivial and cannot be substituted with a default value. For example, code which called the `Action.getResources()` method to gain access to a `MessageResources` object in Struts 1.1.0 were required in 1.2.4 to supply an `HttpServletRequest` object as a new parameter. Some sort of dataflow analysis will be needed to see if the developer has access to that object type in the immediate calling context; if not, the developer needs to either gain access to that type, or excise this dependency from his code.

Change Type, **Supertype Change**, and **Remap** involve a set of conceptually similar transformations. Change Type involves changes to field declaration types, method return types, or the type of one or more parameters in a method's signature. Supertype Change refers to classes whose supertypes changed, thereby altering the fields and methods they inherit. Remap involves systematic replacements of entities with other, similar entities. A simple example of this can be found in the `Attribute` class in jDOM, which merged a `String` and `Element` field into a single `Object` field between versions 1.0.b7 and 1.0.b8. If an API migration tool attempts to alter subclasses of `Attribute` to use the new merged field, simply casting to `String` or `Element` as appropriate, a run-time exception will be thrown.

Add to Interface transformations are cases in which new methods were added to an interface; thus, any client code which implemented those interfaces needed to supply implementations for several new methods.

Extract to Configuration involves changes where entities were deleted because they would later be supplied in configuration files which would be loaded on application startup. This was particularly prevalent in cases where introspection was used to dynamically look up types or methods, such as in Strut's migration to JavaBeans.

Immutable are changes that attempted to make an entity resistant to change, typically by adding the `final` modifier.

Hide are cases in which the visibility of a method or field was reduced, thus restricting the access that callers have to that functionality. Unless this transformation is also accompanied by an Encapsulate transformation, the effect is the same as deleting functionality from the API.

Delete involves cases in which we could not find any replacement for the old functionality in the new version, and thus were forced to conclude that no replacement existed. Deletions were relatively rare; one case in the log4j library saw three classes deleted which were responsible for providing a help dialog option in a Logging graphical user interface that was deemed unnecessary.

Repair Deliberately Damaged Code is a unique case, involving changes made to the `Loader` class in log4j in which a developer deliberately broke the signature of the `loadClass` method by changing its parameter type to a nonsensical value, preventing its use until a bug fix could be developed. The breaking change made it into a public release however, and was only repaired in 1.2.2.[4]

4.3.4 Summary of Transformation Complexity

Table 4 summarizes the kinds of transformations we observed between API versions, with respect to the number of entities being replaced and the number of replacement entities, as described in Section 3.3. The largest number of changes to client code are 1:n in nature (48.1%), with a nearly equal number being 1:1 (47.2%), and the rest comprising n:1 or n:m transformations, and deletions.

		1:1	1:n	n:1	n:m	1:0
Struts	1.0.2–1.1.0	160	58	3	6	4
	1.1.0–1.2.4	71	108	0	0	0
	1.2.4–1.2.6	14	1	0	1	0
	1.2.6–1.2.7	0	17	0	0	0
	1.2.7–1.2.8	0	0	0	0	0
	1.2.8–1.2.9	0	0	0	0	0
log4j	1.0.4–1.1.3	20	19	0	0	1
	1.1.3–1.2.1	26	83	3	0	1
	1.2.1–1.2.2	0	1	0	0	0
	1.2.4–1.2.5	0	0	0	0	3
	1.2.5–1.2.6	0	2	0	0	0
	1.2.7–1.2.8	0	0	0	0	0
	1.2.8–1.2.9	0	1	0	0	0
jDOM	1.0b6–1.0b7	7	10	1	2	2
	1.0b7–1.0b8	12	32	0	0	2
	1.0b8–1.0b9	19	3	0	0	4
totals		329	335	7	9	17

Table 4: Transformation types.

Table 5 summarizes the number of BIs that we classified as involving a fully automatable, partially automatable, or hard-to-automate transformation across all three libraries and their versions. In cases where we determined that multiple transformations had been applied, we rated the complexity of the transformation as that of the most difficult transformation assigned to it. Nearly two-thirds of the needed migration transformations are hard-to-automate, with most of the remainder being fully automatable.

5. PRELIMINARY EXTENSION

We report preliminary results in extending our study from Section 3.2 to utilize the technique of Schäfer et al. [24].

[4] issues.apache.org/bugzilla/show_bug.cgi?id=9305

		F	P	H
Struts	1.0.2–1.1.0	103	19	109
	1.1.0–1.2.4	23	10	146
	1.2.4–1.2.6	14	0	2
	1.2.6–1.2.7	0	0	17
	1.2.7–1.2.8	0	0	0
	1.2.8–1.2.9	0	0	0
log4j	1.0.4–1.1.3	9	4	27
	1.1.3–1.2.1	13	4	96
	1.2.1–1.2.2	0	0	1
	1.2.4–1.2.5	0	0	3
	1.2.5–1.2.6	0	2	0
	1.2.7–1.2.8	0	0	0
	1.2.8–1.2.9	0	0	1
jDOM	1.0b6–1.0b7	7	1	14
	1.0b7–1.0b8	10	8	28
	1.0b8–1.0b9	10	0	16
totals		189	48	460

Table 5: Automatic transformability of BIs. F = fully automatable; P = partially automatable; H = hard to automate.

In essence, their technique examines how client code is migrated between API versions to mine for transformations.

We manually applied this technique for the Struts 1.1.0 to 1.2.4 API migration. For each of the 179 BIs present, we sought test cases for or references to the relevant entities in the test suite associated with Struts 1.1.0, and then compared the results to the test suite in 1.2.4, to see if the source code affected by each BI had been migrated.

We found only two cases where a BI had an associated test case, or was used in another test case, and was migrated to a replacement in the new version of the test suite; both of the migrations matched the changes we had determined in our study. We found a third case in which a BI had already been migrated in the test suite for 1.1.0, in preparation for the eventual change in the API. We note that, since Schäfer et al.'s evaluation was conducted on a single library migration, they would have not captured this migration.

Of the remaining BIs: 21 were on fields and methods that had a unit test associated with their enclosing type, but none of the test cases exercised the binary incompatible entity; 28 were on fields and methods whose enclosing type were referenced in other test cases, but the affected functionality was not exercised; and, 127 BIs had no associated test case, nor were referenced in any other test case.

6. DISCUSSION

Three issues remain to be discussed.

6.1 Recommendation Blindness

In our verification study (Section 3.4), both participants failed to detect wrong change recommendations in cases where a tool/technique had strongly recommended a replacement. We feel this is unlikely to be caused by our participants simply not having examined the code, as a second false change we seeded pointed to nonexistent functionality that both participants detected. We also do not believe that this is due to an inability by our participants to reason about the behaviour and intent of code, as both candidates caught other mistakes with these characteristics, and one participant was able to point out a case in which we were wrong about a replacement. In fact, when it was pointed out to participant P1 after the study that they had missed the case of the erroneous code comment, they were surprised: they remembered checking the replacement, noting "...I guess I was just looking at the code in that area, I wasn't really looking for that [specific] method."

We believe that the participants experienced "recommendation blindness" in their work: a precise and unambiguous recommendation that superficially seemed reasonable was implicitly trusted, and caused the developer to ignore clear evidence in the source code that would have normally caused them to correctly conclude that the recommendation was wrong. We feel this has important implications for recommendation tools in two respects: first, the consequences of false predictions are serious, especially when the tool or technique is presents a high degree of confidence about the recommendation; and second, researchers who analyze only the predictions of their tools as a means of gauging accuracy are at risk of being biased by their tool's predictions.

6.2 Discrepancies with the Literature

In our results (Sections 4.3 and 5), we note discrepancies between some numbers we report, and those reported in Dig and Johnson's study of library evolution [7], as well as both Schäfer et al.'s [24] and Wu et al.'s [26] evaluation of their proposed techniques, with respect to the changes discovered and classified in the Struts 1.1.0 to 1.2.4 migration.

In Dig and Johnson's work [7], a different number of changes are reported as having occurred between the 1.1.0 and 1.2.4 version of Struts. This difference is easily explained by our choice to identify changes based on BIs between the API versions, and to include protected entities in our analysis; Dig and Johnson restricted their study to public entities, and based their analysis on published release notes, as well as a heuristic analysis of the source code.

Dig and Johnson also report that, of the breaking changes discovered in Struts 1.1.0 to 1.2.4, 90% were "refactorable" [7, pp.103]. In contrast, we found that only 12.8% of the BIs in that particular migration were easily performed by mechanical transformation tools; the majority of the changes either require restricted conditions to be met, or some degree of additional specification from a developer that could not be automatically inferred by a tool. To a large degree, this discrepancy is due to how Dig and Johnson classify changes: they are intent on capturing refactorings that alter structure while preserving behaviour, and classify several serious breaking changes (such as deleted class, deleted method, change argument type, changed return type, pushed down method) as being refactorings. While we encountered similar changes, we do not believe that these changes would be easily handled by mechanical transformation tools; instead, the API maintainer, or the client developer, would need to craft some minimal specification that would describe how to remap classes to accommodate these breaking changes. We note that Dig and Johnson also acknowledge this, pointing out in the case of "change return type" that some sort of additional mapping would need to be provided to describe how the new return type mapped to the old [7, pp.96].

In Schäfer et al.'s evaluation of their approach, they indicate that a key assumption is that *"Users* [i.e., client source

code] *of the changed framework exist that are ported to the new version"* [24, p. 477], and they justify this assumption in part by their evaluation of the Struts 1.1.0 to 1.2.4 migration, claiming that *"...the Struts experiment has shown that our approach performs reasonably well, even if no instantiations are available beyond test cases"* [24, p. 477]. However, our results invalidate this justification: the provided test suite addressed barely more than 1% of the BIs caused by the API changes. While Schäfer et al. report 66 transformations, at most two of those involved BIs—the other transformations must either involve non-BI API changes or are false positives, in which client code changed for reasons other than API migration.

In Wu et al.'s evaluation of the AURA tool [26], they note that a feature of their approach is in their tool's ability to detect when functionality (specifically, methods) were deleted between versions of an API, as a means of improving tool accuracy. They presented two evaluations where the effect of accounting for method deletion was compared [26, pp.331,333]; their results for the Struts 1.1.0 to 1.2.4 migration showed that accounting for method deletion raised the relative recall of their tool by 55.9% when compared to prior work. In their investigation, they remarked that 77 methods were deleted between Struts 1.1.0 to 1.2.4 [26, pp.332]. When deleted methods were not accounted for, their approach had lower recall than prior work.

However, in our study we found no cases where an entity was deleted without an appropriate replacement between Struts 1.1.0 and 1.2.4. Finding replacements in this transition is difficult, requiring systematic investigation and extensive reasoning. This highlights the need for a complete and validated set of change data for multiple libraries that undergoes continuous improvement, so that it is clear what changes any novel technique does and does not address.

6.3 Threats to Validity

The most significant threat in our work is to internal validity: our empirical study relies heavily on the correctness of the replacements we discovered for deleted functionality, the appropriateness of the classifications we applied to each change, and the consistency with which we applied the various recommendation techniques. Our verification study provides some evidence that our work is accurate, with only one mistake found in a sample of 20% of the data we evaluated for what was the most difficult library migration to analyze.

We acknowledge that errors may remain in this data; however, the manual nature of the analysis is unavoidable: the change recommendation techniques we applied were not able to find replacement functionality for a BI change in nearly 80% of the cases examined, suggesting that there is currently no single tool or technique which is sufficiently reliable to build such a dataset. Furthermore, there is clearly a need to know how libraries have evolved through investigation that is independent of tools' reported solutions (see Section 6.1); our manual investigation calls into question the reported results of other state-of-the-art tools.

Ideally, test suites would exist or could be created that would capture each BI, and prove that the replacement functionality we determined does in fact work; such a suite would provide tremendous confidence in the accuracy of our results. Sadly, few (if any) test cases existed in the studied libraries that had direct bearing on the BIs we found, meaning that all of these test suites would need to be created.

Our work also suffers from issues of external validity: the Struts, log4J, jDOM, DBCP, and SLF4J APIs are not necessarily representative of all Java libraries, let alone all software libraries in active use. Our goal in this work was to provide an in-depth study of these libraries, to suggest the possible transformations that all libraries may go through, and to provide some empirical comparison as to how well various change recommendation tools work. However, the scalability of studies like this is currently problematic, due to the necessary reliance on manual investigation.

7. CONCLUSION

Library migration remains a challenging proposition. To better understand its nature, we have conducted a detailed, largely manual investigation of the binary incompatibilities between pairs of library versions, to determine what adaptive changes to client code would be needed to migrate to the later version. In addition, we have evaluated a range of change recommendation techniques for their ability to recommend the correct changes.

We found that for any given BI, it was rare that more than one or two change recommendation techniques was able to identify the correct replacement, and on average, a given recommendation or analysis technique was successful for only 20% of the BIs we encountered. We suggest that there may not be a single change-recommendation approach that is generally suitable for all situations; rather, hybrid techniques which incorporate multiple, complementary analysis techniques may prove more effective overall. Recent work has begun to try such approaches [26], but our work here questions those results.

For library migration research, our results are sobering: the majority of API changes we observed could not be automatically transformed in the client code without either severe restrictions on the contexts in which the transformation could be attempted, or some additional specification work by an API maintainer or consumer. This may point to limits on how effective migration techniques based on automatically detecting or capturing refactorings can be.

Finally, we note that the majority of changes to an API were not documented in the associated release notes, nor in the source code comments; attempts to collect change set data from developer documented sources should be careful to supplement that information with other investigative tools.

Our goal is to create a living corpus of change data that not only expands over time with additional systems studied, but is continuously corrected and refined by state-of-the-art research contributions. By inviting other researchers to perform similar kinds of tool-independent change analysis on other software libraries, and contributing their work to this corpus, we hope to increase the validity of this study simply through the sheer volume of complete change data that will hopefully one day exist.[5]

8. ACKNOWLEDGEMENTS

We wish to thank Hamid Baghi, Rylan Cottrell, Soha Makady, Valeh Nasser, and our anonymous reviewers for their helpful comments on this work, and our anonymous participants for their assistance in our validation study. This work was supported by NSERC.

[5]The corpus can be found at lsmr.cs.ucalgary.ca/projects

9. REFERENCES

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

[2] J. Bloch. How to design a good API and why it matters. In *Companion ACM SIGPLAN Conf. Obj.-Oriented Progr. Syst. Lang. Appl.*, pp. 506–507, 2006.

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

[4] B. Cossette and R. Walker. Polylingual dependency analysis using island grammars: A cost versus accuracy evaluation. In *Proc. IEEE Int. Conf. Softw. Mainten.*, pp. 214–223, 2007.

[5] B. Dagenais and M. Robillard. Recommending adaptive changes for framework evolution. *ACM Trans. Softw. Eng. Methodol.*, 20(4):19/1–19/35, 2011.

[6] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *Proc. Europ. Conf. Object-Oriented Progr.*, LNCS 4067, pp. 404–428, 2006.

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

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

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[10] M. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31(2):166–181, 2005.

[11] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Java SE 7 edition, 2012. http://docs.oracle.com/javase/specs/jls/se7/html/index.html.

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

[13] R. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40(10):39–42, 1997.

[14] P. Kapur. Refactoring references for library migration. Master's thesis, Department of Computer Science, University of Calgary, Calgary, Canada, 2010.

[15] P. Kapur, B. Cossette, and R. Walker. Refactoring references for library migration. In *Proc. ACM SIGPLAN Conf. Object-Oriented Progr. Syst. Lang. Appl.*, pp. 726–738, 2010.

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

[17] C. Larman. Protected variation: The importance of being closed. *IEEE Softw.*, 18(3):89–91, 2001.

[18] M. Lehman. Programs, life cycles, and laws of software evolution. *Proc. IEEE*, 68(9):1060–1076, 1980.

[19] B. Morel and P. Alexander. SPARTACAS: Automating component reuse and adaptation. *IEEE Trans. Softw. Eng.*, 30(9):587–600, 2004.

[20] M. Nita and D. Notkin. Using twinning to adapt programs to alternative APIs. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pp. 205–214, 2010.

[21] D. Parnas. Software aging. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pp. 279–287, 1994.

[22] J. Perkins. Automatically generating refactorings to support API evolution. In *Proc. ACM SIGPLAN–SIGSOFT Wkshp. Progr. Analysis Softw. Tools Eng.*, pp. 111–114, 2005.

[23] M. Robillard and R. DeLine. A field study of API learning obstacles. *Empir. Softw. Eng.*, 16(6):703–732, 2011.

[24] T. Schäfer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, pp. 471–480, 2008.

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

[26] W. Wu, Y.-G. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, volume 1, pp. 325–334, 2010.

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