Beyond API Signatures: An Empirical Study on Behavioral Backward Incompatibilities of Java Software Libraries

Eric Ruiz, Shaikh Mostafa, Xiaoyin Wang
Department of Computer Science, University of Texas at San Antonio, TX 78249, USA
{Eric.Ruiz, Shaikh.Mostafa, Xiaoyin.Wang}@utsa.edu

July 17, 2015

Abstract

To make sure that existing client software applications are not broken after a library update, backward compatibility has always been one of the most important requirements during the evolution of software libraries. However, due to various reasons, backward compatibility is seldom fully achieved in practice, so it is important to understand the status, major reasons, and impact of backward incompatibilities in real world software. Previous studies related to this topic mainly focus on API signature changes between consecutive versions of software libraries, while in this paper, we mainly consider behavioral changes of APIs. Specifically, we performed large-scale cross-version regression testing on 68 consecutive version pairs from 15 most popular Java software libraries. Furthermore, we collected and studied 144 real world software bugs caused by backward incompatibilities of software libraries. Our major findings include: (1) more than 1,000 test failures / errors and 280 groups of behavioral backward incompatibilities are detected from 52 of 68 consecutive version pairs; (2) a large portion backward incompatibilities causing real-world bugs are related to user interface, which may be difficult to be detected by current automatic regression testing techniques; (3) the majority of backward incompatibilities are not well documented; and (4) there exists a number of fix patterns for behavioral backward incompatibilities.

1 Introduction

Nowadays, as software products become larger and more complicated, software libraries have become a necessary part of almost any software. For example, a developer may write several lines of code to generate a trivial "Hello World" Android app. When it is executed, it invokes thousand lines of code in software libraries from the Android platform and the underlying Linux system. The

prevalent usage of software libraries has significantly reduced the cost of software development and improved the quality of software products.

Like all other software, software libraries also evolve continuously and frequently to support new features and to improve quality. Since software libraries and their client software are typically maintained by different developers, the asynchronous evolution of software libraries and client software may result in incompatibilities. To avoid incompatibilities, for decades, "backward compatibility" has been well known as a major requirements in the evolution of software libraries. Each API method in an existing version of software library should exactly maintain its behavior in the following versions.

However, in reality, full backward compatibility is seldom achieved, and the resulted incompatibilities have been known to affect software users as well as the success of software projects. For example, Windows Vista is considered to be not very successful, and its failure has been largely ascribed to its backward incompatibility with Windows XP [1]. Also, in October 2014, the automatic system update to Android 5.0 caused a complete dysfunction of SogouInput (the top Android app for Chinese input on mobile phones, with more than 200 million users), which is not patched until 3 days later [2]. Also, a recent study [13] has shown that the usage of instable Android APIs is an important factor affecting the successfulness of Android apps.

With the reported well-known software failures that are relevant to backward incompatibilities, we believe that it is necessary to conduct thorough studies to understand the status and reasons of backward incompatibilities in the real software world, and the result of such studies are able to guide the development more advanced techniques to support detection, documentation, and resolution of backward incompatibilities.

There have been several existing empirical studies [21, 16] on the stability of software libraries, and extensive research efforts on library migration [6, 24]. However, these studies mainly focus on the changes of API signatures (i.e., added, revised, and removed API methods) between consecutive versions of software libraries. Although API signature changes form an important category of backward incompatibilities, they do not describe the whole picture. Even if the signature of an API method remains identical in a new version, it is possible that the behavior of the API method changes (i.e., generate different output or side effect when certain input is fed in). Actually, since API signature changes can be easily detected by compilers, although they may cause compilation errors and extra efforts in library migration, they are less likely to cause real-world bugs, and thus are relatively less harmful compared to API behavioral changes.

To acquire a deeper and more complete understanding of backward incompatibilities in real world software libraries, in this paper, we present an empirical study on behavioral backward incompatibilities on 68 consecutive version pairs from 15 popular Java software libraries. We further inspected 144 real-world bugs that are caused by backward incompatibilities in these libraries. We chose Java software libraries as our subjects for two reasons. First, Java is a very popular programming language, and Java software typically extensively relies on software libraries. Second, most popular Java software libraries are open

source with test code available, so that it allows us to look into backward incompatibilities more deeply in our study.

Specifically, in our study, for each consecutive version pairs, we performed cross-version testing to test the new version of software library with the test code of the old version. After that, we collected the test failures and test errors as backward incompatibilities and inspected them to categorize the reasons behind as well as whether they are well documented. To further investigate the impact of backward incompatibilities on the client software, we collected 144 real world bug reports, categorized the reasons behind these bugs, and studied how these bugs are fixed by either software library developers or client software developers. The major findings of our study include:

- Behavioral backward incompatibilities are very common in popular Java software libraries.
- There are many bug-inducing behavioral backward incompatibilities that are related to user interface and other system components.
- Very small proportion of behavioral backward incompatibilities are documented.
- there exists a number of fix patterns for behavioral backward incompatibilities.

This paper makes three main contributions as follows.

- A large scale experimental study on the existence of behavioral backward incompatibilities in 15 popular Java libraries and 68 version pairs.
- A detailed manual study of 144 real world bugs caused by backward incompatibilities.
- A data set including 280 backward incompatibilities detected in cross-version testing, and 144 real world bugs, serving as a foundation for future research in this area.

We organize the rest of the paper as follows. In Section 2, we introduce the design of our study and the process of data collection. In Section 3, we present the result of our study on both cross-version testing of consecutive versions pairs of software libraries, as well as real-world bugs caused by backward incompatibilities. After that, we discuss the learned lessons and limitations of our study in Section 4, and the related works in Section 5. Before we conclude in Section 7, we indicate a number of future research directions based on the results of our study in Section 6.

2 Study Setup

In this section, we introduce the design and data-collection process of our study. First of all, we give our definition of behavioral backward incompatibilities. *Behavioral Incompatibilities* are backward incompatibilities caused by changing the behavior of a method / field (i.e., changing the default value), while its signature remain untouched. Unlike signature incompatibilities, behavioral incompatibilities are difficult to detect, and the reasons and characteristics of behavior incompatibilities have not been well studied and are still unclear. This is the reason why we try to acquire more understanding of behavioral incompatibilities through studies in this paper.

2.1 Research Questions

In this part of our study, we try to answer the four research questions as follows.

- **RQ1:** How prevalent are behavioral incompatibilities between consecutive version pairs of Java software libraries?
- **RQ2:**What are the bug-inducing backward-incompatibilities and how can they be categorized?
- **RQ3:**What is the documentation status of the incompatibilities causing real-world Backward-Incompatibility Bugs?
- **RQ4:**How are the bug-inducing backward-incompatibilities fixed in real software practice?

With the answers of these questions, we expect to understand, (1) whether backward incompatibility is prevalent in the Java software libraries, and a problem that software developers need to face frequently, (2) whether it is possible to classify behavioral incompatibilities into several categories and develop corresponding detection and resolution techniques, (3) whether backward incompatibilities are documented sufficiently, and what type of technical support may be desired to help documentation of backward incompatibilities, (4) whether there exists certain patterns on fixing backward-incompatibility related bugs.

To answer the four research questions, we designed a study that applies cross-version testing to 68 versions in 15 top Java software libraries, and another bug-report study that involves manual inspection of bugs related to behavioral backward incompatibilities in real world.

2.2 Selection of Version Pairs

Software developers use different levels of versions to mark different granularity of milestones in software evolution. The version strategies and corresponding naming conventions vary a lot in different software projects. Since the code difference between different levels of version pairs varies, the selection of version pairs may have a large impact to our study results.

Table 1: Basic Information of Studied Subjects and Versions

Subject	St. V.	End V.	# V.	St. Time	End Time
OpenJDK	7b157	8b13	2	2011-7	2014-3
Android	4.3.1	5.0.1	2	2013-10	2014-12
log4j	2.0.0	2.1	2	2014-7	2014-10
maven	3.0.0	3.2.5	4	2010-10	2014-12
bukkit	1.2.3	1.7.2	6	2011-12	2013-12
beanutils	1.9.0	1.9.2	1	2008-9	2013-12
codec	1.6	1.7	1	2011-11	2012-9
fileupload	1.2.0	1.3.1	3	2007-2	2014-2
cm-io	2.0	2.4	4	2007-7	2012-4
ela. Search	1.0.3	1.3.9	7	2014-4	2015-2
http-core	4.0.1	4.3.3	6	2009-2	2014-2
jodatime	2.0	2.7	7	2011-5	2015-1
jsoup	1.1.1	1.7.3	10	2010-6	2013-11
neo4j	1.8.3	2.0.3	5	2012-11	2015-2
snakeyaml	1.3	1.11	8	2009-7	2012-9

In our study, we first rule out the alpha and beta versions which are typically immature versions and are not widely used by client software developers. Regarding mature versions, we observe that they mainly fall into two levels. The first level of versions involve major changes in software features and usages, and we refer to them as *Major Versions*. Typically, major versions are developed as separate branches in the version control system, and consecutive pairs (e.g., Java 6 and Java 7) of major versions are maintained simultaneously by software developers . By contrast, the second level of versions are mainly for bug fixes and minor changes inside a major version, and we refer to them as *Minor Versions*. Typically, minor versions corresponds to a certain revision number in the trunk or a branch, and consecutive pairs (e.g., Java6u32 and Java6u40) of minor versions are not maintained simultaneously.

To acquire a full picture, in our study, we study backward incompatibilities both between two consecutive major versions and within a major version. Specifically, if a major version has more than two minor versions, we choose the first minor version and the last minor version to form an inner-major-version version pair. For example, ElasticSearch has four minor versions (1.0.0 through 1.0.3) for major version 1.0, and three minor versions (1.1.0 through 1.1.2) for major version 1.1. So, in our study, we choose four versions (1.0.0, 1.0.3, 1.1.0, 1.1.2), and form 1 major version pairs (1.0.0 to 1.0.3, 1.1.0 to 1.1.2) and two minor version pairs (1.0.3 to 1.1.0).

The details of our selected libraries and versions as presented in Table 1, and the full list of version pairs used in our study are at our project website[1].

2.3 Detection of Backward Incompatibilities

To detect behavioral incompatibilities, we first make sure we can successfully build all the software library versions used in our study, and all test cases pass. Then, we automatically recompile the test code of previous version with the source code of the new version, and iteratively remove the test cases that do

[1] `http://xywang.100871.net/empIncomp.html`

not compile with the new version of source code (typically because they suffer from signature compatibilities), until all test cases can be compiled successfully. Finally, we execute all the remaining test cases and collect test failures and errors.

It should be noted that, one behavioral incompatibility (i.e., behavioral change of an API method) may causes multiple test failures and errors, and we introduce the clustering algorithm for backward incompatibility groups as below.

Clustering of Backward Incompatibility Groups. To cluster test failures and errors in to incompatibility groups, we leverage the observation that, if two test methods do not refer to the same set of methods in the source code of the software library, their failure must not be caused by the same backward incompatibility. It should be noted that, it is still possible that the behaviors of two methods in the software library change due to a same root cause. However, they are typically considered as two backward incompatibilities of the software library. Therefore, in our algorithm, we first cluster all test cases in one test class to a cluster. The reason is that, they may fail due to a same error in the setup method of the test class. After that, for each pair of test classes, if they invoke a same method in the source code of the library, we deem them as interfered. Then, we cluster the test cases based on the closure of the interference relationship. Note that, for JDK, we rule the commonly used methods such as `java.lang` and `java.util`. Note that the way we cluster backward incompatibilities is also conservative, which is consistent with detection process. Therefore, we can guarantee that all of the identified incompatibility groups are truly incompatibility groups.

simply check whether two test cases (failed or with error)

2.4 Collection of Bug Reports

To collect bug reports related to backward incompatibilities. We searched two large on-line open bug repositories: JIRA and Github. Specifically, we used as keywords the combination of terms related to software upgrading (e.g., "upgrade", "update", "version"), and the names of the software libraries listed in Table 1. From the collected bug reports, we *randomly* selected 500 bug reports, and carefully inspected these bug reports. Specifically, we read the developers' comments and other references from the bug report to check whether they are confirmed by the developers to be caused by backward incompatibilities, and retain all the bug reports that are caused by backward incompatibilities.

It should be noted that, when collecting both bugs that are confirmed fixed and the bugs that are decided to be not fixed. The reason is that, backward incompatibility bugs are related to both software libraries and client software, so they can be fixed either at the library side or at the client side. Also, there are cases that a backward incompatibility bug is never fixed because the software library developers refuse to revert their changes, and the client developers did not find a way to work around it. In such cases, the developers may choose to not to migrate to the new version of software library (not possible for runtime

Table 2: Basic Information of Backward-Incompatibility-Related Bugs

Subject	Library Bugs	Client Bugs	Total
Java SDK	9	13	22
Android	15	70	85
Other	31	6	37
Total	55	89	144

libraries such as JDK and Android), or have the users to tolerate the bug (if the bug is relatively minor), or even give up the whole project (we do observe such cases).

With the process above, we collected 144 bugs. We divide these bugs into two groups: *library bugs* that are submitted to the software library project that has backward incompatibilities, and *client bugs* that are submitted to a software client project because it triggers a backward incompatibility of one of its software libraries. The breakdown of collected bugs are shown in Table 2.

From the table, we can observe that, as we used a random selection of bugs, the majority of selected bug reports are from Android and Java SE. The reasons are two fold. First, Java SE and Android are much popular than other software libraries studied. Actually, Java SE is used in all Java projects, and Android is used in 34% of the top 10,000 projects we studied for ranking software libraries, while ApacheHttp which ranks next is used in only 10% of the projects. Second, Java SE and Android are both runtime platforms, so that client software developers do not have control on which version of JVM and Android system will be used with their software. Therefore, backward incompatibilities may be revealed after a Java or Android update at the users' side, and get reported to the client software developers. By contrast, most of the other libraries (e.g., Apache libraries) in our study are packaged with the client software. Thus, client software developers are able to test the backward compatibility of a new software-library version, and work around the backward incompatibilities before the software is released to the users. This actually also explains another observation that, the majority of bugs of Android and Java SE are client bugs, while the majority of bugs of other libraries are library bugs. The reason is that, Android and Java SE backward incompatibilities are more likely to be reported by end users to the client software developers as client bugs, while backward incompatibilities in other libraries are more likely to be reported by client software developers to library software developers as library bugs.

3 Study Results

In this section, we present the result of our study on both cross-version testing of consecutive software-library version pairs, and on real-world bugs caused by backward incompatibilities.

Table 3: Detected Backward Incompatibilities in Software-Library Version Pairs

Sbj.	Failure			Error			B-Incomp.		
	T.	Av.	I/A	T.	Av.	I/A	T.	Av.	I/A
JDK	203	101.5	2/2	15	7.5	2/2	48	24	2/2
and	112	56	2/2	11	5.5	2/2	41	20.5	2/2
log	71	35.5	2/2	0	0	0/2	4	2	2/2
mav	14	3.8	3/4	226	56.5	4/4	25	6.3	4/4
buk	15	2.5	2/6	31	5.2	3/6	8	1.3	4/6
bea	0	0	0/1	0	0	0/1	0	0	0/1
cod	4	4	1/1	6	6	1/1	3	3	1/1
fil	0	0	0/3	12	4	2/3	3	1.5	2/3
cio	4	1	1/4	2	0.5	1/4	3	0.8	2/4
ela	36	5.1	4/7	98	14	3/7	31	4.4	4/7
htt	203	33.8	5/6	15	2.5	4/6	34	5.7	5/6
jod	15	2.2	5/7	6	0.8	2/7	9	1.3	5/7
jso	54	5.4	9/10	2	0.2	1/10	20	2	9/10
neo	5	1	2/5	7	1.4	1/5	5	1	2/5
sna	108	13.5	8/8	14	1.8	4/8	46	5.8	8/8
Tot.	**844**	**12.4**	**46/ 68**	**445**	**6.5**	**28/ 68**	**280**	**4.1**	**52/ 68**

3.1 Backward Incompatibilities of Popular Libraries

We present the detected test failures / errors from software-library consecutive version pairs in Table 3. The first column of the table presents the subject name. To save space, we use only the first 3 letters of each subject. Note that the projects are listed in the same order as in Table 1. The columns 2 to 4 present the total number of test failures detected in all version pairs of a specific subject (abbreviated as $T.$), the average number of test failures detected in each version pair (abbreviated as $Av.$), and the number of versions where test failures are detected (denoted as I) divided by all version pairs of the subject (denoted as A). The columns 5-7 and 8-10 present similar data for test errors and backward incompatibility groups (clustering of backward incompatibility groups is introduced in Section 2.3). It should be noted that test failures and test errors are two different ways by which a test case may fail. Typically, a test failure is raised when an assertion in the test case fails, while a test error is raised when the test throws an unhandled exception or fails to complete (e.g., reaching timeout).

From Table 1, we have two major observations. First of all, we find that behavioral backward incompatibilities are prevalent among these popular Java software libraries. We detect test failures in 13 of 15 subjects, and backward incompatibilities in 14 of 15 subjects. Furthermore, among 68 version pairs we studied, 52 version pairs (76.5%) suffered from backward incompatibilities. Considering that cross-version testing may generate a very low underestimation of the number of incompatibilities, the prevalence of behavioral backward incompatibilities may be even higher than what is shown in the table.

Second, on average, we detected 12.4 test failures, 6.5 test errors, and 4.1 incompatibility groups for each version pair, which shows that one version pair typically suffers from multiple incompatibilities. It should be noted that due to our conservative algorithm to cluster incompatibility groups, the actual number of incompatibilities groups may be even higher.

Table 4: Distribution of Backward Incompatibilities in Different Software-Library Version Pairs

Subject	Total		Average		Incomp. V / All V	
	Mj.	Mn.	Mj.	Mn.	Mj.	Mn.
JDK	35	13	35	13	1/1	1/1
Android	41	N/A	20.5	N/A	2/2	N/A
log4j	3	1	3	1	1/1	1/1
maven	13	12	6.5	6	2/2	2/2
bukkit	3	5	0.8	2.5	3/4	1/2
beanutils	N/A	0	N/A	0	N/A	0/1
codec	3	N/A	3	N/A	1/1	N/A
fileupload	0	3	0	1.5	0/1	2/2
cm-io	3	N/A	0.8	N/A	2/4	N/A
ela.Search	0	31	0	7.8	0/3	4/4
http-core	15	19	5	6.3	2/3	3/3
jodatime	15	N/A	2.2	N/A	5/7	N/A
jsoup	13	7	2.2	1.8	7/7	3/4
neo4j	5	0	1.7	0	2/3	0/2
snakeyaml	46	N/A	4.6	N/A	10/10	N/A
Total	**195**	**91**	**4.1**	**4.3**	**36/47**	**16/21**

Documentation of Backward Incompatibilities. Since in our experiment, we guarantee that all tests pass in their original version, the test errors and failures detected should be mainly caused by intended behavioral changes. So we furthered studied the documentation status of these changes. Specifically, we randomly chose 50 test failures / errors[2] from 50 different incompatibility groups (10 from JDK, 10 from Android, and 30 from other libraries), and then carefully checked the corresponding release notes, API documents, and migration guides (for Android). We discovered that only 17 (6 of 10 in Java, 5 of 10 in Android, and 6 of 30 in other libraries) of the 50 test failures / errors have their behavioral changes been documented.

Distribution of Incompatibilities in major / minor versions. Beyond the overall status of backward incompatibilities between consecutive version pairs of software libraries, we further studied the difference between major and minor version pairs. The results are presented in Table 4.

From Table 4, we have the observation that, both types of version pairs suffered from about 4 backward incompatibilities on average, and about 75% to 80% of each types of version pairs are backward incompatible.

We notice that, to rule out signature-level incompatibilities, we delete the test cases that do not compile in our cross-version testing. Since major version pairs have more signature-level incompatibilities, less test cases are being used in cross-version testing. Therefore, the similar numbers in the table may not imply similar severity of backward incompatibility. However, the table does show that there are many backward incompatibilities between minor version pairs, which is not a good news, because minor version pairs are typically supposed to be used for bug fixing and minor changes, and to be backward compatible.

[2]The list of studied test failures / errors is in our project web site.

3.2 Bugs Related to Backward Incompatibilities

In the previous subsection, we investigated the prevalence of backward incompatibilities among popular software libraries, by detecting test errors / failures from cross-version testing. In this subsection, to answer the second research question, we present the study result on real world bug reports related to backward incompatibilities, and explore how backward incompatibilities are affecting the client software developers. The basic information of the collected bug reports are presented in Table 2. In the following subsections, we first identify bug reports that are related to signature incompatibilities, and cross-software duplicate bug reports. Then, we categorize bug-inducing behavior incompatibilities by their incompatible behavior and the condition to invoke the incompatible behavior, as well as study the documentation status of them. Finally, we study how these bug are fixed or resolved by library developers and client developers.

3.2.1 Signature Incompatibilities

Since the bug reports are randomly collected, some of the bug reports are related to signature-level incompatibilities. As we mentioned in Section 1, since signature-level incompatibilities fail compilation, they are not likely to result in real world bugs.

Among the 144 bug reports, we do find 18 of then related to signature-level incompatibilities. Specifically, 6 of the 18 bugs are library bugs, which are reported by client developers to request of the recovery of a removed API method. Other 6 of the 18 bugs are runtime errors from client software running on Android, Java, and Bukkit. Because these libraries serve as runtime environments, when automatic updates happen, the software running on them will throw exceptions such as "NoSuchFieldException" or "UnSupportedOperationExceptio".

The rest 6 of the 18 bug reports are more interesting. As for these bugs, the developers of relevant client software use reflections to invoke the API methods with signature changes, or use "instanceof" operations, or downcasts on the classes whose inheritance hierarchy has changed. Thus the signature incompatibilities become latent, and cannot be detected by compilers. It should be noted that, 4 of the 6 bug reports are related to Android. Reflection is encouraged in Android to address backward incompatibilities (i.e., call different API methods for different Android versions), so it seems that reflection serves as a double-edge sword here.

3.2.2 Behavioral Incompatibilities

Besides, the 18 bug reports discussed above, the rest 126 bug reports are all related to behavioral incompatibilities. Before we perform more in-depth investigation, we first manually scanned the bug reports to detect *cross-software duplicate bug reports*. Inner-software duplicate bug reports are typically labeled and we did not select them when collecting our bug report set. However, different client software may fail due to a same backward incompatibility, so the

Table 5: Basic Information of Backward-Incompatibility-Related Bugs

Subject	Library Bugs	Client Bugs	Total
JDK	8	10	18
Android	13	51	64
Other	29	1	30
Total	50	62	112

relevant client bug reports are "duplicate" with each other. After the duplicate-bug-report detection, we identified 112 incompatibilities as presented in Table 5.

3.2.3 Incompatible Behaviors

With investigation of the bug-inducing behavioral incompatibilities, we identify the following major types of incompatible behaviors.

Unhandled Exception indicates that, in the new version of the software library, an API method throw a exception that it does not throw under the same usage scenario in the old version.

Infinite Loop indicates that, in the new version of the software library, an API method will fall in an infinite loop and never return under certain usage scenario.

Return Variable Change indicates that the return value of the API method changes in the new version under certain usage scenario. Specifically, we divide this category into three sub-categories. A *Value Change* indicates that a primitive value (e.g., integer, boolean, String) is changed. The value can be of the return variable itself or of a field of the return variable. A *Type Change* indicates that the actual type of the return value is changed, although the signature itself remain unchanged. This typically happens when the return type in the API method signature has many subtypes (e.g., `java.lang.Object`). A *Structure Change* indicates that, no primitive values in the return object is changed, but the object is organized differently (i.e., pointer fields or sub-fields of the object is changed).

Besides the control flow and the return value, a backward incompatible behavior may be related to changes of other parts of the software itself and the system (generally categorized as **Other Effects**), such as side effect on other variables in memory (*Memory*), changing the user interface (*User Interface*), the file system (*File Sys.*), and the way system events are sent and received (*Sys. Event*). An example of *Sys. Event* is *Bug-408:"be able to configure as a default SMS app in KitKat"* in `TextSecure`.

In Android 4.4, SMS apps are no longer able to send SMS to the SMS provider (rejected silently) in the Android system, unless they are reset to receive a broadcast `SMS_DELIVER_ACTION`.

The breakdown of all incompatibilities according to the backward incompatible behaviors is presented in Table 6. In the table, column 1 presents incompatible behaviors. Columns 2 and 3 present the number of library bugs (denoted as

Table 6: Categorization of Incompatible Behavior

Behavior		Android		JDK		Other		All
		L	C	L	C	L	C	
Unhandled Exception		2	21	5	5	13	1	47
Infinite Loop		2	2	0	0	0	0	4
Ret. Var. Change	Value Change	2	2	1	3	9	0	17
	Type Change	1	2	1	0	1	0	4
	Structure Change	0	0	0	1	3	0	4
Other Effects	Memory	1	0	1	0	1	0	3
	User Interface	5	19	0	1	1	0	26
	File Sys.	0	2	0	0	1	0	3
	Sys. Event	0	1	1	0	0	0	2

L) and client bugs (denoted as C) from Android. Columns 4-7 present similar data for Java and Other subjects. Column 8 presents the total of each line.

From Table 6, we have the following observations.

First of all, unhandled exception and infinite loop account for 51 of the 112 (45%) incompatible behaviors. Although it is possible that these behaviors are more likely to be reported as bugs due to their severity, this proportion of 45% is not very far from the proportion of test errors among test errors / failures (35% that can be calculated from numbers in Table 3). This implies that simpler and more scalable techniques that targets at unhandled-exception related changes in the software library may be quite useful in detecting backward incompatibilities.

Second, primitive value changes accounts for 17 incompatible behaviors (15 %). This actually implies that, symbolic summarization of API methods is still useful, but may be not necessary to detect the majority of backward incompatibilities.

Third, user interface changes accounts for 26 incompatible behaviors (22.3%), which is the second largest category. We investigated the bug reports in this category and discover that, most bugs are related to UI settings, such as theme colors and padding widths. For example, in *Bug-46:"Bold ZeroTopPadding-TextView displays cut off on 4.4"* of `Android-Betterpickers`, the developers found out that, on Android 4.4, if they do not update the padding settings before showing the date information (which was what they did on Android 4.3), only half of the date information can be seen. It should be noted that, user interface bugs are not just decoration problems, and they may largely affect software usages (i.e., information cannot be seen, or buttons go outside the screen and cannot be clicked).

3.2.4 Invocation Constraints

In our study, we further investigated the conditions that invoke bug-inducing behavioral incompatibilities, and identified the following major types of conditions.

Always indicates that the incompatible behavior always happens as long as the corresponding API method is invoked. Such incompatibilities can be easily detected by regression unit testing, so they are more likely to be intended behavioral changes. However, it should be noted that such incompatibilities may

Table 7: Categorization of Invocation Conditions

Conditions		Android		Java		Other		All
		L	C	L	C	L	C	
Always		5	14	2	1	2	0	24
Environment		0	1	0	0	1	0	2
Special Type		0	2	0	0	2	0	4
Multiple APIs		4	21	1	5	7	1	39
Input	Trivial Value	0	0	1	0	2	0	3
	String Format	2	2	3	3	6	0	16
	Specific Field	0	4	0	0	0	0	4
	Specific Value	2	7	1	1	9	0	20

not be easily detected in the client software, because the relevant API method may not easily invoked, and the incompatible behavior (e.g., changed return value) may be overwritten and thus become unseen under certain conditions.

Environment indicates that the incompatible behavior happens only under certain environments (e.g., operating systems, language settings).

Special Type indicates the argument must be of a specific subtype of its parameter type in the API method signature.

API indicates that a number of other API methods must be invoked before the backward incompatible API method to invoke its incompatible behavior.

Input indicates that the incompatible behavior happens when a certain input value is fed into the corresponding method. Specifically, we divide this category into four sub-categories. *Trivial Value* indicates that a null pointer or an empty string / list is the invoking input. *String Format* indicates that specific structured strings are the invoking input. *Specific Field* indicates that objects with specific values at a specific field of the input object are the invoking input. *Specific Field* indicates that specific primitive values (not including strings) are the invoking input.

The breakdown of bug reports according to incompatible-invoking conditions is presented in Table 7. From the table, we have the following observations.

First of all, 24 of 112 (21%) incompatibilities always happen, causing at least 15 client-side bugs. As we mentioned above that such incompatibilities tend to be intended, this actually calls for a better documentation of the behavior changes. We will discuss on this topic later in the study of documentation status of backward incompatibilities.

Second, only 2 of the incompatibilities are related to the environment. This may be largely due to the platform independence of Java, so we doubt whether this conclusion can be generalized to other programming languages.

Third, 39 incompatibilities (35%) occur only after certain other API methods are invoked. This actually implies that a lot of behavior incompatibilities happen under special usage scenarios. For library developers, they may not be able to come up with such usage scenario easily, but client software code may be helpful.

Table 8: Documentation Status of Incompatibilities

Behavior		Android		Java		Other		All
		L	C	L	C	L	C	
No Doc.		12	43	7	6	29	1	98
Doc.	Release Notes	1	1	1	3	0	0	6
	JavaDoc	0	3	0	1	0	0	4
	Migration Guide	0	4	0	0	0	0	4

3.2.5 Call Backs

One interesting finding in the 112 studied incompatibilities is that, there are 6 of them related to call backs. All of these 6 incompatibilities are from Android bugs (4 client bugs and 2 library bugs). One example of call-back-related incompatibilities is *Bug-62100: "WebViewClient.onPageFinished() called multiple times"* of Android system. In Android 4.4, this call back method is called multiple times when there are multiple frames in the web view, while it is called only once before 4.4. Such change may cause severe problem, if the client developers close some resources (closing a closed resource may cause exceptions) or change some global objects such as counters, in the call back.

Call-back backward incompatibilities can easily happen, because library developers may simply change the way they are calling a certain method inside their code, without knowing that this method is being overridden by client developers. Also, call-back backward incompatibilities are difficult to avoid, because library developers cannot make any assumptions on the content of the call back.

3.3 Documentation of Bug-Inducing Incompatibilities

To answer the third research question, we further studied the documentation status of the bug-inducing incompatibilities. Since these incompatibilities are bug inducing, we predict that they may be poorer documented then the incompatibilities detected from cross-version testing (34% documented), and the results shown in Table 8 confirm our guess. In Table 8, the first column presents whether the documentation status (and the place if documented). The rest columns are organized similar to Table 6.

From Table 8, we have the following observations. First of all, bug-inducing behavioral incompatibilities are very poorly documented. Only 14 (13%) incompatibilities are documented. Also, the documented changes are relatively scattered, especially for Android (in release notes, JavaDocs, and Migration guides).

Second, we notice 8 client bugs of Android and 4 client bugs of Java are related to documented behavioral changes. This imply that we may need a better way then documentation to convey the information of behavioral change and remind client software developers about such changes.

Table 9: Resolution of Library Bugs

Resolution		Android	Java	Other	All
Fixed	Reverted	1	0	2	3
	Patched	7	5	18	30
	Double Support	0	0	1	1
Not Fixed	Intended	6	2	10	18
	Discouraged	1	2	0	3

3.4 Resolution of Backward-Incompatibility Bugs

To answer the fourth research question, we further studied how the real world bugs related to backward incompatibilities are resolved (note that they may be not fixed). It should be noted that, we include the bug reports that are caused by signature incompatibilities and duplicate bug reports in the study. The reason is that, signature-incompatibility-related bugs are typically resolved similarly to behavior-incompatibility-related bugs. Also, cross-software duplicate bugs may be fixed different in different client software. We present and discuss the resolution of library bugs and client bugs in the following two subsections, respectively.

3.4.1 Resolution of Library Bugs

The breakdown of bugs according to how they are resolved is shown in Table 9. The first two columns present the types of resolution. If a library bug is fixed, we check whether it is fixed by a simple revert of the previous change, a patch of the previous change, or library developers decided to support both the previous behavior and the new behavior (typically by adding a parameter, and set either the previous behavior or the new behavior as default). If a library bug is not fixed, we study how library developers response to the bug report, and check whether it is intended behavior, or the developer is reporting a behavioral change on internal APIs which should not be used by client developers.

From Table 9, we have the following observations. First, 21 of 55 library bugs are not fixed. The major reason is that it is an intended behavior. It should be noted that, since these behaviors are reported as library bugs, they are likely to already cause some bugs or at least test failures at the client side.

Second, among the 34 bugs that are fixed, most of them are patched, which shows that the incompatibilities are due to side effect of other productive changes.

3.4.2 Resolution of Client Bugs

The breakdown of bugs according to how they are resolved is shown in Table 10. The first two columns present the types of resolution. If a client bug is fixed, we check whether it is fixed by (1) changing the incompatible API; (2) changing the input value; (3) adding an API invocation to set a certain internal-state field before or after the incompatible API invocation; (4) converting the return value of the incompatible API invocation to the original value; (5) a global structural code change; (6) updating libraries; (7) changing configuration of

Table 10: Fixes of Client Bugs

Resolution		Android	Java	Other	All
Fixed	Change API	4	2	3	9
	Change Input	13	2	1	16
	Add Set Field	17	1	0	18
	Return Convert	6	0	0	6
	Structural	11	6	0	17
	Config	2	0	0	2
	Lib. Update	2	0	1	3
	Bypass	4	0	0	4
Not Fixed	Wait Lib. Fix	4	2	0	6
	Tolerate	7	0	1	8

software; or (8) bypassing the incompatibility behavior by skipping software features. An example of bypassing is the resolution *Bug-969: "Android 5.0 crash when trying to open the app"*, in which the client developer simply commented out the backward-incompatible resource fetching code that causes the crash.

For the client bugs that are not fixed, we discovered two resolutions. The first resolution is that the client developer simply decided to wait until a new version library is released. One reason of such resolution is that, the incompatibility is caused by a regression bug, so the client developer waits for the library developers to release a bug-free version. Another reason (and the major reason in our study) is that, the incompatibility affects a third-party library that the client developers are relying on. Since the client developers cannot change the code of the third-party library (sometime they even do not have access to the source code), they are not able to resolve the incompatibility, and have to wait for the new version of the third party library.

The second solution is that, the client developer simply tolerate the behavior change (if the incompatibility does not cause crashes). They may simply ask their users to get used to the new behavior such as a UI change, or transfer the incompatibility to downstream developers.

From Table 9, we have the following observations.

First of all, 14 client bugs are not fixed. It should be noted that, we find that most developers are willing to and have tried to fix the bugs, but backward-incompatibility bugs are relatively more difficult to fix, because they typically involve code written by other people.

Second, although there are 17 client bugs fixed through structural changes, which may be very complicated. 49 client bugs are fixed through changing API, changing input value, add an API to set field, or convert the return value to the original value. Such resolutions are relatively simple have the potential to be automated.

Reporting to Library Developers. In our study, we further studied whether client developers would like to report their bugs to the library developers. Among the 89 client bug reports, we find that the symptom is reported to library developers in only 6 bug reports. In most of the cases, the developers simply search through the Internet to find a workaround. Also, for the 6 reported bugs, only 3 are fixed by the library developers, while the other 3 are rejected because the corresponding backward incompatibilities are intended

behaviors.

4 Discussion

In this section, we discuss about the lessons learned from our study, as well our limitations and the threats to our study.

4.1 Lessons Learned

Prevalence of Behavioral Incompatibilities. Our study shows that behavioral incompatibilities are very common in popular Java software libraries. Although cross-version testing is able to reveal only a small portion of potential backward incompatibilities, we still detected more than 1000 test failures and errors, as well as identified 280 incompatibility groups. We also find that behavioral incompatibilities do cause lots of real bugs in the real world.

Detection of Behavioral Incompatibilities. Our study shows that, the invocation conditions of behavioral incompatibilities vary, and the invocation of other APIs is one of the major condition. This implies that testing an API method together with other methods that may change relevant internal memory state may benefit the detection of behavioral incompatibilities. We also find that, user interface change is one of the major symptom of behavioral incompatibilities. This calls for automatic user-interface checking techniques to support regression testing of GUI applications.

Documentation of Behavioral Incompatibilities. Our study shows that the documentation status of behavioral incompatibilities is very poor. Even if a behavioral change is documented, there are still many relevant client bugs. We believe that, advanced techniques on documentation of behavioral incompatibilities is in a great need and will help reduce many bugs related to backward incompatibilities. The technique should be able to document various factors of behavioral changes such as the APIs that help to invoke behavioral incompatibilities, and the changes on the user interface.

Resolution of Behavioral Incompatibilities. Our study shows that, there are a number of code patterns for fixing bugs related to behavioral incompatibilities. Specifically, a lot of bugs are fixed through direct adjustment of the input values, or conversion of the return values. Also, we find that many bugs are fixed by directly setting proper values to a field whose value is changed due to the behavioral change of the API method. The existence of such patterns show possibilities that many bugs related to behavioral incompatibilities can be fixed automatically.

4.2 Limitations and Threats

Limitations. As an early step towards better understanding of behavioral backward incompatibilities, our study has a number of limitations.

First of all, we use cross-version testing to detect behavioral backward incompatibilities in software libraries. This result in an under-estimation of the number of behavioral incompatibilities between version pairs. Also, since we require all test cases pass in their original versions, the detected incompatibilities are biased to the intended behavioral changes (since the relevant test cases are already fixed). However, the major goal of our study on regression testing is to show the prevalence of behavioral backward incompatibilities, and we believe the above mentioned limitations do not affect our conclusion.

Second, when studying incompatibility-related bugs, we search the bug repositories with keywords such as "update". It should be noted that bugs related to incompatibilities do not have very obvious keywords, such as "deadlock" for concurrency bugs. In particular, some backward-incompatibility related bugs may stay in the software for a long time, and the client developers may not realize the root cause of the bug even after it is fixed. Thus, our selected bugs may be biased to those bugs that are easily found to be relevant to backward incompatibilities.

Third, in our study on bug reports, we largely depend on the comments and code commits of developers to determine whether a bug is fixed or not, and the relevant code locations. It is possible that developers make mistakes and thus affect the precision of our results.

Fourth, in our study of documentation status, we carefully checked release notes, migration guides, and API JavaDocs. However, it is still possible that library developers are documenting the behavioral changes elsewhere, such as in bug repositories. Thus, we may miss some documented backward incompatibilities. However, we believe that the places that we check are also the places client developers may refer to. If the behavioral changes is documented somewhere hard to find, it is still not well documented.

Threats to Validity The major threats to internal validity of our study is the potential errors and mistakes in the process of building software and performing regression testing, studying the bugs, and doing the statistics. To reduce this threat, we carefully wrote all the tools we used, and checked the results for correctness. The major threats to external validity is that, our conclusion may hold for only Java software libraries, and the libraries under study. Furthermore, our conclusion may hold for only the 144 bugs studied. To reduce this threat, we chose the most popular and thus representative Java software libraries, as well as randomly chose the bugs to be studied.

5 Related Works

The major research topics that are related to our research are: studies on the stability of software libraries, summarization of library changes, and migration for library evolution.

5.1 Studies on the Evolution of Software Libraries

Researchers have noticed that software libraries are evolving frequently for a long time, so a number of studies have been conducted on the evolution of software libraries. Raemaekers et al. [21] proposed a measurement of software-library stability which considers API method difference and code difference, and studied the stability of 140 industrial Java systems based on the measurement. McDonnell et al. [16] studied the stability of Android APIs (in terms of added and removed classes and methods), and the time lag between the release of API changes and the corresponding adaptation at the client software side. Espinha et al. [7] interviewed 6 web client software developers and conducted an empirical study on four widely used web services to understand their API evolution trends, including the frequency of API changes, and the time given client developers to upgrade to the new version of services. Bavota et al. [3, 4], studied the evolution of software dependency upgrades in the apache software ecosystem. The authors analyzes the factors that affect developers' decision on software library dependency upgrades and the impact of the upgrades. The existing research efforts mainly focus on signature-level API changes (Raemaekers et al.'s work considers the amount of code difference on top of API signature changes) to measure API changes and stability. By contrast, our study focuses on behavioral changes of software libraries, which are more difficult to be detected and may cause more severe consequences.

There have also been a number of research efforts on the impact of software-library to client software. Linares-Vsquez et al. [13] further studied the relationship between change proneness of APIs methods and the successfulness of client software that uses those API methods in Android software ecosystem. Bavota et al. [5] further extends the work with more detailed experimental results. These research efforts shows that backward incompatibilities have much effect on the successfulness of client software, and thus motivate the study in our paper.

5.2 Summarizing Changes of Software Libraries

There have also been research efforts trying to summarize changes between two consecutive versions of a software library. On the signature level, Wu et al. [23] proposed AUCA, an auditor for API changes, that reports a large variety of signature-level changes of APIs. Moreno et al. [18] proposed ARENA, an automatic tool to summarize software-library changes and generate release notes. Specifically, ARENA analyzes source code changes of a software library and generate a natural-language-based notes summarizing library changes changes including additions and removals of files, classes, methods, changes of method signatures, etc.

On the behavior level, summarizing the behavior of a method from its source code has been a hot topic for a long time in the area of compositional program analysis [8, 22]. There are also some research efforts on change-aware behavior summarization, that summarize the behavioral changes between two versions of a method. Specifically, McCamant and Ernst [15, 14] proposed to represent

behavior API methods with program invariants generated with Daikon, and to represent behavioral changes as violations of program invariants. More recently, Person et al. [19, 20] proposed differential symbolic execution to summarize as symbolic expressions of inputs the semantic difference between two versions of a method. Lahiri et. al [12] proposed SymDiff, a tool that leverages a modularized approach to check semantic equivalence of different code versions, and calculate program paths that can reveal code behavioral difference.

These proposed techniques can handle general behavioral changes of methods, while our study actually shows that a large portion of real-world behavioral incompatibilities are related to user interface, execution environment, etc., which raise new challenges for these techniques. Also, our study shows that behavioral incompatibilities follow some patterns, so that more specific and more scalable techniques may be developed accordingly.

5.3 Support for Library Migration

Another research topic closely relevant to our work is support for library migration, including the mapping of APIs between two consecutive versions of a software library and inference of usage patterns of newly added APIs. Godfrey and Zou [9] proposed a number of heuristics based on text similarity, call dependency, and other code metrics, to infer evolution rules of software libraries. Later on, S. Kim et al. [11] further improved their approach to achieve fully automation. M. Kim et al. [10] inspected existing framework evolution process to gather a number name-changing patterns and used these patterns to infer rules of framework evolution. Dagenais and Robillard [6] proposed *SemDiff*, which infers rules of framework evolution via analyzing and mining the code changes in the software library itself. Wu et al. developed *AURA* [24], which further involves multiple rounds of iteration applying call-dependency and text-similarity based heuristics on the code of software library itself. Most recently, Meng et al. [17] proposed *Hima*, which further enhances *AURA* by involving information from comments of code commits between two consecutive versions of software libraries.

It should be noted that, almost all existing support for library migration are at the API signature level (i.e., providing replacement API methods to make client code compile). While these approach can largely enhance the productivity of library migration, they may not be able to handle API methods with behavioral changes, which can go through compilation and become runtime errors in the later stages of software life cycle.

6 Future Works

In the future, we plan to further explore the following research directions.

First of all, our study focuses on Java software libraries, so our conclusion may not be generalized to other programming languages. Therefore, we plan to conduct similar studies on software libraries written in other languages, es-

pecially non-object-oriented languages to confirm or extend our conclusion. We also plan to inspect more backward-incompatibility-related bug reports.

Second, as we mentioned in Section 2, regression testing with developers' test suite may find only a small portion behavioral incompatibilities, and thus results in a very course underestimation of the number of behavioral incompatibilities. In the future, we plan to leverage automatic test generation and more advanced automatic test oracles to better detect behavioral backward incompatibilities.

Third, due to the difference in the popularity of API methods, the potential influence of backward incompatibilities varies. A backward incompatibility is more important if the relevant API method is used (directly or indirectly) more widely. We plan to further study the influence of behavioral incompatibilities and signature incompatibilities.

Fourth, in our study, we find a number of challenges and research opportunities including behavioral incompatibilities related to reflections, call backs, GUI, and execution environments, better documentation of behavioral incompatibilities, etc. We plan to address some of these challenges in the future.

7 Conclusion

In this paper, we present a study on behavioral backward incompatibilities based on regression testing of 68 version pairs of 15 Java software libraries, and inspection of 144 real world bugs. From our study, we find that behavioral backward incompatibilities are prevalent among Java software libraries, and caused most of real-world backward-incompatibility bugs. Furthermore, most of the behavioral backward incompatibilities are expected by developers of software libraries, but are rarely well documented. We also categorize behavioral backward incompatibilities according to the incompatible behaviors and invocation conditions, and discussed the challenges of using existing techniques to detect and fix them.

References

[1] Criticism of windows vista. `https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en`. Accessed: 2014-08-30.

[2] Sougou. `https://play.google.com/store/apps/details?id=com.sohu.inputmethod.sogou&hl=en`. Accessed: 2014-08-30.

[3] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. The evolution of project inter-dependencies in a software ecosystem: The case

of apache. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 280–289, 2013.

[4] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, pages 1–43, 2014.

[5] G. Bavota, M. Linares-Vasquez, C. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *Software Engineering, IEEE Transactions on*, (99):1–1, 2014.

[6] B. Dagenais and M. P. Robillard. Recommending adaptive changes for framework evolution. In *Proc. ICSE*, pages 481–490, 2008.

[7] T. Espinha, A. Zaidman, and H.-G. Gross. Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 84–93, 2014.

[8] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali. Compositional may-must program analysis: Unleashing the power of alternation. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 43–56, 2010.

[9] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, February 2005.

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

[11] S. Kim, K. Pan, and E. J. Whitehead, Jr. When functions change their names: Automatic detection of origin relationships. In *Proc. WCRE*, pages 143–152, 2005.

[12] S. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Reblo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification*, pages 712–717, 2012.

[13] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, 2013.

[14] S. McCamant and M. Ernst. Early identification of incompatibilities in multi-component upgrades. In *European Conference on Object-Oriented Programming*, pages 440–464, 2004.

[15] S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 287–296, 2003.

[16] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 70–79, 2013.

[17] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 353–363, Piscataway, NJ, USA, 2012. IEEE Press.

[18] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora. Automatic generation of release notes. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 484–495, 2014.

[19] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 226–237, 2008.

[20] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 504–515, 2011.

[21] S. Raemaekers, A. van Deursen, and J. Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387, 2012.

[22] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 187–206, 1999.

[23] W. Wu, B. Adams, Y.-G. Gueheneuc, and G. Antoniol. Acua: Api change and usage auditor. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 89–94, 2014.

[24] W. Wu, Y. Guéhéneuc, G. Antoniol, and M. Kim. AURA: A hybrid approach to identify framework evolution. In *Proc. ICSE*, pages 325–334, 2010.