Broken Promises: An Empirical Study into Evolution Problems in Java Programs Caused by Library Upgrades

Jens Dietrich
School of Engineering and Advanced Technology
Massey University
Palmerston North, New Zealand
j.b.dietrich@massey.ac.nz

Kamil Jezek
Faculty of Applied Sciences
University of West Bohemia
Univerzitni 8, 306 14 Pilsen
Czech Republic
kjezek@ntis.zcu.cz

Premek Brada
Faculty of Applied Sciences
University of West Bohemia
Univerzitni 8, 306 14 Pilsen
Czech Republic
brada@kiv.zcu.cz

***Abstract*—It has become common practice to build programs by using libraries. While the benefits of reuse are well known, an often overlooked risk are system runtime failures due to API changes in libraries that evolve independently. Traditionally, the consistency between a program and the libraries it uses is checked at build time when the entire system is compiled and tested. However, the trend towards partially upgrading systems by redeploying only evolved library versions results in situations where these crucial verification steps are skipped. For Java programs, partial upgrades create additional interesting problems as the compiler and the virtual machine use different rule sets to enforce contracts between the providers and the consumers of APIs.**

We have studied the extent of the problem on the qualitas corpus, a data set consisting of Java open-source programs widely used in empirical studies. In this paper, we describe the study and report its key findings. We found that the above mentioned issues do occur in practice, albeit not on a wide scale.

I. Introduction

Re-use has long been seen as an important approach to reduce the cost and increase the quality of software systems [34],[8]. One of the re-use success stories is the use of libraries (jar files) in Java programs [18], [32]. This is facilitated by a combination of social factors such as the existence of vibrant open source communities and commercial product eco-systems, and Java language features like name spaces (packages), class loading, the jar meta data format and the availability of interface types. In particular, open source libraries such as *ant*, *junit* and *hibernate* are widely used in Java programs.

However, the way libraries are used is changing. It used to be common practice to import fixed versions of used libraries into a project and then to build and distribute the project with these libraries. This meant that the Java compiler and additional tools like unit testing frameworks were available to check the overall product for type consistency and functional correctness at the source code level. Newer build tools like Maven and Gradle have replaced references to fixed versions of libraries with a mechanism where a symbolic reference to a library, often restricted by version ranges, is used and then resolved against a central repository where libraries are kept. However, integration still happens at build time, safeguarded by compilation and automated regression testing.

Driven by the needs for systems with high availability ("24/7 systems") and product lines, a different way to deploy and integrate systems has become popular in recent years: to swap individual libraries and application components at runtime. This feature can be used to replace service providers and perform "hot upgrades" of 3rd party libraries. In Java, this is made possible by its ability to load and unload classes at runtime [24, ch.5.3.2],[17, ch. 12.7]. The most successful implementation of this idea to date is OSGi [30], a dynamic component framework that uses libraries wrapped as components ("bundles") with a private class loader. References between bundles are established through a process called *wiring* at runtime by the OSGi container. This approach is now widely used in enterprise software, including application server technology (WebLogic, WebSphere) and development tools (the Eclipse ecosystem).

This has created some interesting challenges for maintaining application consistency. OSGi allows users to upgrade individual libraries at runtime by installing their component jar files and re-loading the respective classes. This mechanism circumvents the checks done by the compiler and delegates the responsibility to establish consistency between referencing and referenced code to the Java Virtual Machine (JVM). In many cases, this does not really matter as the rules of *binary compatibility* used by JVM at link time are similar to the *source compatibility* rules checked by the compiler. However, there are some subtle differences that can lead to unexpected runtime behaviour. For instance, signature changes like specialising method return types are compatible according to the rules used by the compiler, but incompatible from the JVMs point of view. Java features like erasure and constant inlining also contribute to this mismatch. The Java documentation emphasises that "these problems cannot be prevented or solved directly because they arise out of inconsistencies between dynamically loaded packages that are not under the control of a single system administrator and so cannot be addressed by current configuration management techniques" [27].

978-1-4799-3752-3/14/$31.00 © 2014 IEEE

A commonly used solution is to add another layer of constraints to restrict linking. For instance, OSGi-based systems should use semantic versioning [29]. Its rules for matching the versions of exported packages of bundles with the version ranges of packages imported by other bundles can be used to restrict the use of classes across bundles. This however delegates the responsibility to the programmer who has to assign the versions to the components, leading to even more difficult issues [4]: *a*) Many code changes are very subtle (as discussed below) and it can not be expected that all programmers understand the effects of seemingly minor API changes. *b*) Programmers have to precisely follow the rules of the versioning scheme and its semantics (preferably formal ones). *c*) The users of a library need to have the same understanding of the versioning scheme as its producer. Therefore, any approach that relies on manually assigned versions is inherently error-prone.

The objective of this paper is to investigate the mismatch between Java compilation- and link-time consistency rules, and to study to what degree these issues occur in practice. The text is organised as follows. We review related work in section 2, and classify the evolution problems we investigate in section 3. Section 4 describes the methodology used to set up and execute the experiments. The results of these experiments are reported in section 5, followed by a discussion of threats to validity in section 6 and the conclusion in section 7.

II. Related Work

The notion of binary compatibility goes back to Forman et al [16], who investigated this phenomena in the context of IBM's SOM object model. In the context of Java, binary compatibility is defined in the Java Language Specification [17, ch. 13]. Drossopoulou et al [15] have proposed a formal model of binary compatibility. A comprehensive catalogue of binary compatibility problems in Java has been provided by des Rivières [10]. The problems we have investigated here are a subset of this catalogue.

There is a significant body of research on how to deal with evolution problems in general, and how to avoid binary incompatibility in Java programs in particular. For instance, Dmitriev [14] has proposed to use binary compatibility checks into an optimised build system that minimises the number of compilations. Barr and Eisenbach [1] have developed a rule-based tool to compare library versions in order to detect changes that cause binary compatibility problems. This is then used in their Distributed Java Version Control System (DJVCS), a system that helps developers to release safe upgrades. Binary component adaptation (BCA) [21] is based on the idea to manipulate class definitions at runtime to overcome certain binary compatibility problems. Dig et al [13] and Savga and Rudolf [9] have proposed a refactoring-based solution to generate a compatibility layer that ensures binary compatibility when referenced libraries evolve. Corwin [6] has proposed a modular framework that adds a higher level API on top of the Java classpath architecture. This approach is similar to OSGi, a framework that is now widely used in industry.

To the best of our knowledge there are no comprehensive empirical studies to assess the extent of the problem caused by binary evolution issues. Dig and Johnson [12] have conducted a case study on how APIs evolve on five real world systems (*struts*, *eclipse*, *jhotdraw*, *log4j* and a commercial mortgage application). They found that the majority of API breaking changes were caused by refactoring, as responsibility is shifted between classes (e.g., methods or fields move around) and collaboration protocols are changed (e.g., renaming or changing method signatures). Their definition of API breaking changes does not distinguish between source and binary compatibility ("a breaking change is not backwards compatible. It would cause an application built with an older version of the component to fail under a newer version. If the problem is immediately visible, the application fails to *compile or link*" [12]). Mens et al [26] have studied the evolution of Eclipse from version 1.0 to version 3.3. Eclipse is of particular interest to us as it is based on OSGi and therefore supports dynamic library upgrades through its bundle / plugin mechanism. The focus of this study was not on API compatibility but to investigate the applicability of Lehmann's laws of software evolution [23]. However, they found significant changes (i.e., additions, modifications and deletions) in the respective source code files. It can be assumed that many of these changes would have caused binary compatibility issues if the respective bundles had evolved in isolation. Cosette and Walker have studied API evolution on a set of five Java open source programs [7]. The focus of their work was to assess change recommendation techniques that can be used to give developers advise on how to refactor client code in order to adapt to API changes. They investigated binary incompatibility issues between versions of the programs in their data set, and found numerous incompatibilities for three programs in their data set (*struts*, *log4j*, *jdom*), and no incompatibilities for the other two programs (*slf4j*, *dbcp*). Two of these libraries (*struts* and *log4j*) are also part of the data set we are using.

Our analysis of the use of OSGi semantic versioning [2] has demonstrated that in current open-source Java projects, the versions assigned to bundles often do not reflect real changes; we have however not checked yet whether this problem actually impacts system compositions. In our previous work, we have also investigated the problem that occurs when integration builds are skipped in plugin-based components [11]. We have argued to integrate unit testing into runtime composition, and have shown how several errors and contract violations can be detected in Eclipse with this approach.

We have used the qualitas corpus data set [33] in our study. This data set has been widely used in empirical studies, including several studies on how APIs and libraries are used [31], [18], [32]. Lämmel et al [22] have investigated API usage in Java programs using an alternative corpus based on the Source Forge code repository. The focus of these studies was to establish the level and the characteristics of reuse in Java programs. None of these studies has investigated API compatibility issues that are the result of library evolution.

III. Evolution Problems

There are several types of system consistency problems that result from (source or binary) incompatibilities between evolving libraries and the programs using them. *Binary incompatible* changes result in errors or unexpected behaviour when an existing program is executed with a changed version of a library. The notion of binary compatibility is defined in [17, ch. 13]. In most cases, the problem is detected during linking and results in a linkage error. However, there are some cases where the impact is more subtle and leads to changes of the program behaviour. *Source incompatible* changes result in compilation errors as defined in [17] when a program is recompiled with the changed version of a library it depends on.

A comprehensive catalogue of evolution problems is presented by des Rivières [10]. We use this collection as a starting point and in this section define several categories of evolution changes grouped by the type of incompatibility issues they cause. For any program version pair, a non-zero number of changes of the given kind is counted as only one change occurrence (e.g. for library X version v, regardless of the number of methods removed from X_v's classes there is only one change in the removed methods category).

A. Binary and Source Incompatible Changes

The first type of evolution problems is caused by changes to libraries that are neither source nor binary compatible. This means that in general *refactoring* of the client program is necessary to re-establish compatibility between the program and the new version of the library it uses.

In practice however, the necessity of refactoring depends on which part of the library is actually being used. For instance, if a class is removed from a library and this class isn't referenced by a program, then the respective change is contextually compatible [3] – is non-breaking only for this particular program and may still be an incompatible change for another program using the given library.

This group includes several categories where artefacts (packages, classes, methods and fields) are removed. Artefact renaming is considered as the removal of the artefact with the old name followed by the addition of an artefact with the new name. We do not distinguish between methods and constructors (the latter are treated as methods returning `void`)[1].

- C1 Removal of a type (class, interface).
- C2 Incompatible type change. This category combines two kinds of changes: (a) type signature changes like removal of implemented interfaces or changed type parameters, (b) structural changes of the type as defined by the following categories.
- F1 Removal of a field.
- F2 Incompatible field type change. Type compatibility is defined in (C2), when generic types are used, their erasure is evaluated.
- M1 Removal of a method. This is defined as the absence of a method with the same name and arity. The removal of overloaded methods is captured in the next category (M2).
- M2 Incompatible method change. This is defined by comparing the return type and the parameter types for methods with the same name and the same number of parameters; the types are evaluated as defined in (C2).
- MOD Modifier changes. Incompatible modifier changes are: non-final to final, non-abstract to abstract, or using more restrictive access rights. This applies to methods, fields and classes.

The reason for the dependency (non-orthogonality) of C2 on other categories is the granularity of change and data interpretation – C1 and C2 together give a clear single point of reference for determining whether the given library version has evolved in a problematic way, at the type level.

B. Binary Incompatible but Source Compatible Changes

The second group of evolution problems consists of changes that are binary incompatible but source compatible. It is this kind of problem that occurs only when the mode of deployment is changed from integration builds to partial library upgrades, for instance when systems have been (re)modularized for better reconfiguration and evolution. These problems cause runtime (linkage) errors, but can easily be solved through recompilation.

The list presented in this subsection is not mutually exclusive with the changes described in III-A. Some of the categories in this group are special cases of a more general category defined in III-A. The remainder of this section develops a list of these changes, starting with the example in Listing 1.

```
// library version 1
public class Foo {
  public static java.util.Collection foo() {return null;}
}
// library version 2
public class Foo {
  public static java.util.List foo() {return null;}
}
// client program using the library
public class Main {
  public static void main(String[] args) {
       java.util.Collection list = Foo.foo();
       System.out.println(list);
  }
}
```

Listing 1. A source compatible but binary incompatible evolution

It is easy to see that this evolution is source compatible: the old return type is replaced by its subtype. The rules of API evolution at the source code level are similar to the rules of safe subtyping (aka Liskov Substitution Principle [25]) – to honour the contract between the client program and the (old) API method, preconditions must not be strengthened and postconditions must not be weakened [10]. But this is clearly the case here: returning a `List` instead of its supertype `Collection` actually strengthens the postcondition. The compiler will apply this reasoning and compilation of the program with version 2 of the library succeeds.

However, the situation changes when version 2 of the library is built independently and the program is executed with the

[1] In Java byte code, constructors are represented as methods with the reserved name `<init>`.

upgraded library. The JVM tries to resolve the method reference with a method "with the name and descriptor specified by the method reference" [24, ch. 5.4.3.3]. When inspecting the byte code of `Main`, the method is referenced by the descriptor `foo()Ljava/util/Collection;`. The changed return type changes this descriptor, and linking fails with a linkage error (more precisely, a `NoSuchMethodError`). The situation is similar when parameter types are generalised - this can be seen as weakening preconditions. From the users point of view these errors occur during program execution. I.e., they are likely to be perceived as program (and not platform) errors.

A specific case are overridden methods. The problem is that methods cannot be overridden by methods with either specialized or generalized parameters. For this reason, such changes are not source compatible if the respective methods are overridden in the program using the respective library. More specifically, if the `@Override` annotation is used, compilation fails when the signature of the overridden method changes. If the annotation is not used, the method will simply be added because the compiler does not consider this as overriding. This can lead to *unintentional overloading*, a practice known for creating errors that are difficult to trace.

For method return types, the compiler checks for consistency between the return types of the overriding and the overridden method. Complete equality is not required here, as Java supports covariant return types [17, ch.8.4.8]. However, when the return type of a method in a library is specialised, there is no guarantee that the return type of an overriding method is still a subtype of the new return type of the overridden method. Therefore in general, this change is (binary and source) incompatible as well. To deal with the issues caused by overriding methods, we use the term *used-only method* to refer to a method declared in a library, and invoked but neither overridden nor implemented in the client program that uses this library. All *final* methods are automatically used-only, for non final methods, the used-only property depends on the usage context.

There are also (rare) situations when generalising parameter types can lead to compilation errors in case the respective method is overloaded. Then the compiler tries to choose the most specific method [17, ch.15.12]. If this fails (i.e., if the method invocation is ambiguous), a compilation error occurs. We therefore define the following two categories:

M.R1 The return type of a used-only method is replaced by one of its subtypes.

M.P1 A parameter type of a used-only method is replaced by one of its supertypes.

The next set of change categories deals with primitive method return and parameter types. Widening the return type of a method breaks source compatibility as it forces clients to use explicit casts. On the other hand, narrowing return types is source compatible. Replacing a primitive parameter or return type by its associated reference type ("wrapper class") or vice versa is binary incompatible but source compatible as the compiler applies autoboxing and auto unboxing, respectively [17, ch. 5.1]. There are two permitted combinations of conversions and boxing/unboxing [17, ch. 5.3]: boxing followed by replacing the wrapper type by one of its super types ("widening reference conversion"), and unboxing followed by a widening primitive conversion. For instance, if a client invokes a method with the signature `foo(int)`, a signature change to `foo(java.lang.Object)` is binary incompatible but source compatible. We therefore define the following incompatible change categories for primitive vs. wrapper types:

M.R2 The primitive return type of a used-only method has been narrowed.

M.R3 The primitive return type of a used-only method has been replaced by its wrapper type.

M.R4 The return type of a used-only method was a wrapper type and has been replaced by its associated primitive type.

M.P2 A primitive parameter type of a used-only method has been widened.

M.P3 A primitive parameter type of a used-only method has been replaced by its wrapper type or one of its supertypes.

M.P4 A primitive parameter type of a used-only method was a wrapper type and has been replaced by its associated primitive type or a type wider than this primitive type.

The JVM does not generate linkage errors when the checked exceptions declared by a method are changed. In particular, source compatible changes (removing an exception from the list of declared exceptions or replacing a declared exception by one of its subclasses) are also binary compatible.

There are only few changes to field types that are source compatible. Both wrapping and unwrapping are source compatible but not binary compatible. Specialising reference field types and narrowing primitive field types breaks source compatibility with clients that write these fields and is therefore only source compatible for fields declared as final. Generalising reference field types or widening primitive field types breaks compatibility with clients that read these fields.

The field-related incompatibilities are:

F3 The primitive field type has been replaced by its wrapper type.

F4 A wrapper type used as the field type has been replaced by its associated primitive type.

F5 A final primitive field type has been narrowed.

F6 A final field type has been replaced by one of its subtypes.

C. Constant Inlining

Finally, constant inlining and folding can also cause problems that can be fixed through recompilation. The Java compiler uses an optimisation for constants (static final fields) that are either primitives or strings. Instead of referencing these fields, the values are copied into the (byte code of the) referencing class. When the value changes in a subsequent version of the library, the value is not updated unless the program is re-compiled against the new version of this library. The Java compiler can also simplify certain expressions for these types (folding) in order to inline them.

According to our definition above, this is a change that is not binary but source compatible. However, according to the definition in the Java Language Specification [17, ch. 13.2] binary compatibility is defined as "linking without error". Technically, this is the case – a redefined constant does not cause linkage errors. But such a change is potentially more dangerous because the use of incorrect constant values can cause application errors that are more subtle and difficult to fix than explicit linkage errors.

We therefore define the following category:

INL The value of a static final field that has either a primitive or String type is changed.

D. Binary Compatible but Source Incompatible Changes

The examples discussed so far seem to suggest that binary compatibility implies source code compatibility. However, this is not the case. There are examples of binary compatible changes that are not source compatible. For instance, consider the code in Listing 2. At runtime, the parameter type of the `List` generic type is ignored when the byte code is checked for binary compatibility. Since the `size()` method does not refer to the generic parameter type, the byte code does not contain a `checkcast` instruction. Therefore the upgrade succeeds. On the other hand, recompiling the project with library version 2 fails because of the type mismatch between `List<String>` and `List<Integer>`.

```
// library version 1
public class Foo {
      public static java.util.List<String> foo() {
            return new java.util.ArrayList<String>();
      }
}
// library version 2
public class Foo {
      public static java.util.List<Integer> foo() {
            return new java.util.ArrayList<Integer>();
      }
}
// client program using the library
public class Main {
      public static void main(String[] args) {
            java.util.List<String> list = Foo.foo();
            System.out.println(list.size());
      }
}
```

Listing 2. A binary compatible but source incompatible evolution

Another example is adding checked exceptions to a method or generalising already declared checked exception types. While these changes do not lead to linkage errors, they are generally not source compatible.

We believe that these cases are very rare, and represent *accidental technical debt* – the program keeps on working but some refactoring may be required eventually when an integration build is performed. We have therefore excluded these changes from this study.

IV. Methodology

In this section, we briefly describe the set up of the experiments described in the rest of the paper. The purpose of these experiments is to find out how frequent the evolution problems described above occur in real-world programs, and to measure the impact these changes have on *actual* client programs.

A. Data Set

We have studied the API evolution on the Java programs in the qualitas corpus [33]. The qualitas corpus is a comprehensive, curated set of Java programs which contains both their binaries and source forms. The current version (20120401) contains 111 programs. The full release (20120401f) combines the standard release (20120401r) with the evolution release (20120401e) which contains multiple versions of programs, a total of 661 versions. The qualitas corpus has been widely used in empirical studies, including several studies investigating APIs usage [31], [18], [32].

For the purpose of our study, we removed two programs from the data set: *eclipse* and *azureus*. Both are plugin-based and rely heavily on custom class loaders, making static analysis difficult and error-prone. This resulted in a data set containing 109 programs and 564 program versions.

B. Ordering Program Versions

We were interested in the relationship between a particular version of a program and its direct successor. For this purpose, we created a list of project versions that explicitly defines their order. In many cases the order is obvious, as common versioning schemes such as `major.minor.micro` imply a linear order. However, some projects use certain tokens such as *beta*, *rc* (release candidate), *cr* (candidate release), *ga* (general availability) and *sp* (service pack). By manually checking their evolution succession we were able to take project-specific versioning schemes like [19] into account.

C. Cross-Referencing Projects

We analysed all programs in the corpus for occurrences of changes between versions that are not compatible. We then also looked for evidence whether this had an impact on other programs using the libraries implemented and exported by the projects. For this purpose, we cross-referenced programs in the corpus as follows.

1) For each program version, we extracted sets of *program libraries* (provided by the project) and *third party libraries* (used by the project). This was done by recursively searching the respective project version folders for jar files, and classifying files with names containing the project name as program libraries, and all other jar files as third party libraries.
2) In a second step, third party libraries used in projects were matched to program libraries in the project defining these libraries. This was done by comparing the (binary) content of the respective libraries. We did not compare just names because often libraries are renamed by the projects using them, e.g. by stripping or modifying version information.
3) In the last step, we removed dependencies on libraries which are used as part of the build process but not referenced in program code; this is common for libraries like *ant*, *antlr* and *pmd* used for automated code generation

TABLE I
POPULAR PROGRAM VERSIONS REFERENCED BY OTHER PROGRAMS IN THE CORPUS

library	version	incoming relationships
ant	1.5.1	5
ant	1.5.3.1	20
ant	1.6.2	10
ant	1.6.5	25
ant	1.7.1	13
antlr	2.7.2	4
antlr	2.7.4	6
antlr	2.7.6	45
junit	3.8.1	24
junit	3.8.2	4
junit	4.4	2
junit	4.7	5

or quality control. To find out whether this was the case, we constructed a program dependency graph from the byte code using DependencyFinder[2] and checked this graph for edges linking classes in the program with classes in this library.

The above process resulted in a set of 212 dependencies between program versions. Figure 1 shows the programs in the corpus and their dependencies. We use the term *relationship* for a reference between a particular version of a program and a particular version of another program it uses. The most popular programs used by other programs are `ant` (81 relationships), `antlr` (61 relationships) and `junit` (35 relationships). Most relationships (132) have some version of *hibernate* as the source. This is due to the fact that the evolution edition of the corpus contains 100 different versions of hibernate (from version 0.8.1 to version 4.1.0). Table I shows selected program versions and the number of other program versions in the corpus using them as third party libraries.

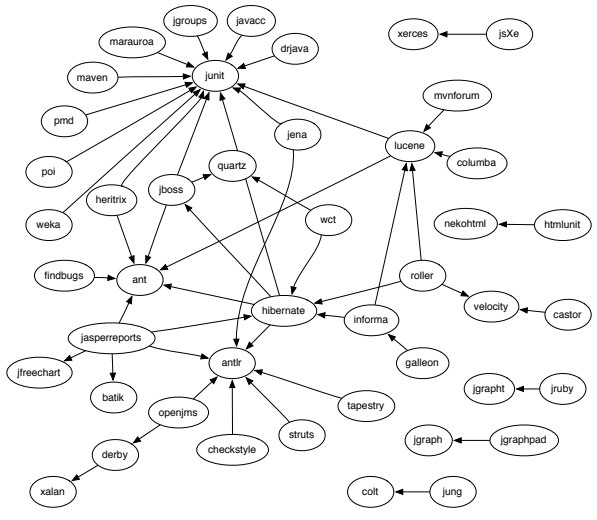

Fig. 1. Cross-project Dependencies between Programs in the Qualitas Corpus

[2]http://depfind.sourceforge.net/

D. Code Compatibility Analysis

We used mainly byte code analysis tools to investigate the compatibility between projects, in particular ASM [5] and JaCC [20]. The latter is a tool specifically developed for API compatibility analysis, it also uses ASM but provides a higher level of abstraction. To analyse the use of inlinable constant definitions, we had to analyse source code to find references that are removed from byte code by the compiler. We developed some ad-hoc scripts for this purpose that use simple regular expression matching.

V. RESULTS

In this section, we present the results of various experiments conducted to investigate to which extent the change categories discussed in Section III occur in practice. These experiments are based on the techniques described above and use the qualitas corpus data set.

A. Program Distribution Completeness

In a first set of experiments we have checked the corpus projects for completeness of the distribution. I.e., we investigated whether all classes referenced in the program are defined either in the program itself or in one of the libraries distributed with the respective program. We detected several cases where this was not the case.

These cases fall into two categories. Firstly, there are cases where referenced classes are defined in third-party libraries that are not included in the project distribution. For instance, there are projects with references to *junit* types (24 in total) where the *junit* library is not part of the program. The reason might be that *junit* is usually provided by development environments like Eclipse as a “system" library and is therefore not explicitly included as third party library. *junit* is, however, not an isolated case. We found 8659 referenced classes that were not part of the respective distribution among the projects in the corpus.

Secondly, we found a significant number of projects (36) referencing classes in `com.sun.*` packages (973 such references in total). We also found 21 projects using `sun.*` packages (73 references). This includes some popular programs such as *ant*, *antlr* and *junit*. The use of these packages implies that the given programs are not easily portable to alternative JVM implementations not containing these classes in their standard libraries.

B. Binary and Source Incompatible Changes

We have investigated 455 version pairs representing “atomic” upgrades, i.e., adjacent versions of the same program according to the version order described in Section IV. The goal was to detect changes of program libraries which could potentially cause source incompatibility in client applications, requiring refactoring.

We found that 344 such potential version upgrades are incompatible and only 111 upgrades are compatible. In other words, 75% of upgrades are not compatible and the number of incompatible upgrades is about 3 times higher than compatible upgrades. Note that these results are not normalised, even a

TABLE II
DISTRIBUTION OF BINARY AND SOURCE INCOMPATIBLE API CHANGES BY CATEGORY

category	junit	hiber-nate	lucene	ant	antlr	all
C1	18	71	22	15	12	289
C2	19	86	25	17	15	331
F1	13	59	18	14	11	251
F2	8	37	21	7	10	203
M1	19	86	22	16	13	314
M2	17	79	25	13	10	296
MOD	11	71	21	9	9	240

TABLE III
NUMBER OF COMPATIBLE VERSION CHANGES THAT ARE API STABLE FOR SOME POPULAR LIBRARIES

change type	junit	lucene	ant	hibernate	antlr
all	22	27	40	99	19
compatible	3	2	3	10	4

single incompatibility issue is enough to classify a version upgrade as incompatible. Details are shown in Table II for some frequently used programs. This table shows the number of versions of the respective program where at least one evolution problem from any of the categories defined above was detected.

We also investigated how many programs in the corpus had completely stable APIs. I.e., we were interested in programs with no adjacent version pairs exhibiting the incompatibility problems defined in Section III-A. We only included programs with more than four versions because API stability in programs with only a few versions might occur accidentally and their data would not be meaningful. This way, only 14 programs were included: *ant* (21 versions), *antlr* (20), *argouml* (16), *colt* (5), *freecol* (28), *freemind* (16), *hibernate* (100), *jgraph* (39), *jhotdraw* (6), *jmeter* (20), *jung* (23), *junit* (23), *lucene* (28) and *weka* (55).

Interestingly, out of this set, only two programs have completely stable APIs: *freecol* and *jmeter*. On the other hand, the most popular programs are riddled with incompatible changes as shown in Table III.

C. Binary Incompatible but Source Compatible Changes

In this experiment, we looked for incompatible changes that can be addressed by recompilation as defined in Section III-B. These changes are relatively infrequent compared to the change categories defined in Section III-A. Table IV provides a summary of the results which are based on the 455 version pairs studied.

We also checked whether any methods included in these results are overridden in other programs in the corpus. These was not the case for any of these methods, i.e. we can consider these methods as used-only as defined in Section III-B.

This section concludes with some examples of incompatibilities found. Each listing starts with the change category as defined in Section III-A, followed by the two versions of the library that were compared, the class (type) name and the name and the arity of the method. Method parameter and return types are listed using the following syntax to describe changes: `<old type> -> <new type>`.

Example M.R1, M.P1 hibernate-3.6.0 → hibernate-3.6.1

```
Class: org.hibernate.criterion.Property
Method: eq(1)
 Return type:
  org.hibernate.criterion.Criterion
  -> org.hibernate.criterion.Simple-
  Expression
 Argument:
  org.hibernate.criterion.Detached-
  Criteria -> java.lang.Object
```

Example M.P3, M.P1 poi-2.5.1 → poi-3.6:

```
Class: org.apache.poi.hpsf.wellknown.
PropertyIDMap
Method: put(2)
 Argument: int -> java.lang.Object
 Argument: java.lang.String ->
 java.lang.Object
```

Example M.R3, M.P3 freemind-0.8.1 → freemind-0.9.0:

```
Class: freemind.modes.StylePattern
Method: getEdgeWidth(0)
 Return type: int -> java.lang.Integer
Method: setEdgeWidth(1)
 Argument: int -> java.lang.Integer
```

Example M.R4, M.P4 derby-10.1.1.0 → derby-10.6.1.0:

```
Class: org.apache.derby.catalog.
IndexDescriptor
Method: getKeyColumnPosition(1)
 Return type: java.lang.Integer -> int
 Argument: java.lang.Integer -> int
```

D. Impact of Incompatible Changes

In the next experiment, we checked whether any of the incompatible API changes observed had an actual impact on other programs in the corpus. I.e., we investigated how many *potential problems* analysed in the previous two subsections would turn into *real problems* if the respective programs employed a dynamic library update mechanism.

For this purpose, we used the 212 dependencies between cross-referenced project versions as described in IV-C. We

TABLE IV
DISTRIBUTION OF BINARY INCOMPATIBLE BUT SOURCE COMPATIBLE API CHANGES BY CATEGORY

category	junit	hiber-nate	lucene	ant	antlr	all
M.P1	2	18	7	3	2	78
M.P2	0	0	2	0	0	17
M.P3	0	2	0	0	0	4
M.P4	0	0	0	0	0	2
M.R1	1	16	2	0	0	42
M.R2	0	0	0	0	0	4
M.R3	0	1	0	0	0	4
M.R4	0	0	0	0	0	1
F3 - F6	0	0	0	0	0	0

then replaced the actually referenced program version by all successive versions available in the corpus. We ignored the dependencies that could not be resolved within the project – these are the dependencies described in Section V-A. We only investigated references to the most widely used programs in the corpus according to the results in Section IV-C: *hibernate*, *lucene*, *ant*, *junit*, and *antlr*. This resulted in 5866 dependencies between programs and versions of libraries used by the respective programs.

Then we analysed how many of these dependencies are incompatible; an incompatible dependency change is detected if a program P uses a library L_1, this library is evolved to a later version L_2, and an incompatible change is detected in a class, method or field defined in L_2 and *actually used* in P. We detected numerous binary incompatible dependency changes from the categories defined in Section III-A. Below we provide their total counts, i.e. the numbers of cases of library versions that are potential updates (newer versions) but cannot be actually used because of binary incompatibilities.

There are 204 dependencies which exhibit compatibility problems due to incompatible types (C2), in 31 cases referenced fields were removed (F1), 45 libraries contain incompatible field changes (F2), in 171 cases referenced methods were removed (M1), and 145 times newer methods are incompatible (M2). No errors were detected for removed types (C1) or incompatible modifiers (MOD). These numbers are grouped by all libraries for simplicity, while the next paragraphs discuss significant aspects behind them.

It is interesting to compare these numbers with those in Table II (last column) – for example, 61% of the potential problems due to incompatible type changes become real issues when partially upgrading the respective programs, and similarly 22% for field changes; on the other hand, none of the type removals (C1) turns into a real problem due to the (lack of) their actual use.

Although we found numerous library compatibility violations, they were detected to affect only 8 client program versions. Namely, *columba-1.0* is incompatible with 22 *lucene* versions, *informa-0.6.5* with 22 *lucene* versions, *informa-0.7.0* with 27 *hibernate* versions and 22 *lucene* versions, *jasperreports-3.7.3* with 18 *hibernate* versions, *mvnforum-1.0* with 31 *lucene* versions, *mvnforum-1.2.2* with 9 *lucene* versions, *roller-2.1.1* with 18 *hibernate* and 22 *lucene* versions and *roller-4.0.1* with 22 *lucene* versions.

The above results show that from a client program's perspective, changes of commonly used APIs can have a significant impact because code has to be refactored in order to use the newer versions. Many compatibility problems detected in our study arise from the use of *lucene* and *hibernate*. An example is the renaming of the method `delete` in `org.apache.lucene.index.FilterIndexReader` to `deleteDocument` somewhere between *lucene* versions 1.4.3 and 1.9.1[3].

[3]Intermediate versions of *lucene* are neither part of the corpus nor available from the lucene homepage.

In addition, this analysis indicates that the current API versioning practice does not provide much help in determining compatible changes. For example, a common compatibility problem when using *hibernate* is the usage of `NullableType` as the type of constants defined in `org.hibernate.Hibernate`. These constants were marked as deprecated in version 3.5.3, and their type was changed to `BasicType` in 3.6.0. This shows that incompatible API changes do occur when only the micro or minor version number of a program changes.

We also investigated a number of incompatible changes that can be solved simply by recompiling the program with the new version of the respective library. This is very rare: we found only two examples (in the projects that do not belong to the group of the most popular ones) where this applies.

The first case was detected in *jaspereports*. Version *jaspereports-1.1.0* is compatible with *jfreechart-1.0.1*, but the following minor change in *jfreechart-1.0.13* causes a binary (only) incompatibility in the M.P1 category:

Example M.P1 jfreechart-1.0.1 → jfreechart-1.0.13:

```
Class: org.jfree.data.time.TimeSeries
Constructor: TimeSeries(2)
 Argument: java.lang.String
  -> java.lang.Comparable
 Argument: ...
```

The second case occurs in the *galleon* project. Version *galleon-2.3.0* uses *informa-0.6.5* but is not compatible with *informa-0.7.0-alpha2*[4]. The incompatibility is caused by a specialised method return type (M.R1):

Example M.R1 informa-0.6.5 → informa-0.7.0-alpha2

```
Class: de.nava.informa.core.ChannelIF
Method: getItems(0)
 Return type: java.util.Collection
  -> java.util.Set
```

Note that compatibility issues related to method signature changes (MR.* categories) only apply to used-only methods. We therefore checked whether any method detected in this category was overridden in any of the projects using the respective library. While there are a total of 512 cases where methods defined in one corpus program are overridden in another corpus program, none of these methods was classified in any of the MR.* categories.

E. Constant Inlining

Incompatible evolution related to constant inlining occurs frequently. We have found 4222 cases where the definition of a constant (either its type or its value) is changed between two adjacent versions. However, these records originated from a small number of programs (10 programs, 9% of the programs in the corpus): *hibernate* (3025), *antlr* (761), *argouml* (231),

[4]The "alpha" version probably marks an unstable version, but this is the last version of this particular project available.

lucene (93), *jgraph* (37), *freemind* (26), *hsqldb* (22), *weka* (11), *ant* (11) and *jfreechart* (5).

We have found only two constant type changes (both in *lucene*, from `Integer` to `String` and `String` to `char`, respectively). We detected 4744 cases where constants were removed. This includes cases where the respective package or type defining the constant was removed or renamed. Again, these changes are in a small number of projects: *antlr* (1312), *hibernate* (1310), *weka* (553), *argouml* (463), *hsqldb* (439), *lucene* (289), *jhotdraw* (112), *tomcat* (109), *ant* (84), *jung* (37), *colt* (17), *freemind* (15), *checkstyle* (3) and *jgraph* (1).

While the number of incompatible value changes seems to be dramatic, a closer inspection reveals that most of these constants are used in parser code. Often, parser APIs are regenerated as part of the build process, and generated integer values are used as constant values. The respective classes often use certain naming patterns, such as `*Lexer`, `*Parser`, `*TokenType` and `*ParserConstants`. These constants are not intended for public use. Removing constants defined in classes with names matching these patterns reduces the number of incompatible changes dramatically to 252 occurrences.

The next pattern we have observed is the use of constants to define library versions. Classes containing version constants include `antlr.Version`,`org.antlr.Tool`, `freemind.main.FreeMind`, `freemind.main.FreeMindApplet`, `net.sf.hibernate.cfg.Environment`, `org.hibernate.cfg.Environment` and `org.jgraph.JGraph`. Keeping versions numbers in code duplicates the information in library meta data (manifests), and can lead to problems when applications rely on reasoning about this version at runtime. One such scenario is when an application loads a service provider class at runtime that is only available in a certain library version, and guards this with a reference to the version. Inlining would then invalidate this guard condition. Removing constants representing version information leaves only 123 incompatible value changes.

We have also investigated references to constant definitions that were just about to change, i.e. when the respective constant value in a given library changed in its next version. We found only a single example where this happens: the class `org.jgraph.pad.actionsbase.lazy.HelpAbout` defined in *jgraphpad-5.10.0.2* references `org.jgraph.JGraph.VERSION` defined in *jgraph-5.9.2.0* and changed in *jgraph-5.9.2.1*.

A particularly interesting case is the removal of a constant referenced in client code. In the corpus, there are seven references to constants defined in *hibernate* referenced by the *springframework* library (versions 1.1.5, 1.2.7) that were removed – the Spring types `org.springframework.orm.hibernate.LocalSessionFactoryBean` and `LocalSessionFactoryBeanTests` reference `CONNECTION_PROVIDER` and `TRANSACTION_MANAGER_STRATEGY`. These constants are defined in `net.sf.hibernate.cfg.Environment` in *hibernate-2.1.8*, but removed in *hibernate-3.0*. This is caused by a major refactoring in the hibernate code base when all packages changes the prefix from `net.sf.hibernate` to `org.hibernate`.

VI. Threats to Validity

There are certain threats to the validity of this study. Firstly, we have studied only programs from a corpus of open source projects; the situation might be different for commercial applications. Secondly, our study focused on the compatibility of method invocations as seen by the compiler and the JVM. With the exception of the study on constant inlining, we ignore semantic issues – i.e., whether a new version of a method could throw unchecked exceptions or simply change the way numerical values are calculated. This is a significantly harder problem to study. Thirdly, our study might contain a few false positives in the M.P1 category due to the issue described in Section III-B when the compiler fails to select the most specific method. Finally, our method to cross-reference programs using provided and used libraries described in IV-C is relatively crude. We may have missed some instances of library usage if a used library has been repackaged (and not only renamed). While this is not common practice, it is still possible, for instance when a program is deployed using a single jar file that includes all packages from the libraries it uses. We also miss references to libraries that are resolved at build time against central repositories. These references can occur when newer build tools like *maven* or *gradle* are used.

VII. Conclusions

In this paper, we have investigated the question whether moving from integration builds to partial library upgrades introduces a new category of errors into Java programs. The short answer is that there are a lot of potential issues but only a few actual errors in the qualitas corpus we studied.

After analysing 109 programs (each with several versions) with 212 program dependencies, we found that the *potential* for problems caused by unguarded evolution is high – as much as 75% of all version upgrades are not compatible. However, we only detected very few *actual* problems: only 8 concrete client program versions are affected by incompatible changes in the libraries these programs use. This might be due to the low number of program dependencies in the corpus as well as to the low level of library classes usage in client programs.

Despite not finding massive incompatibility problems, we did detect some potentially dangerous practices. We believe that applying some simple practices can significantly reduce the number of binary incompatibilities. Firstly, developers should use package naming to distinguish between public and private parts (e.g. `internal` packages in many OSGi projects). This allows verification tools to detect the illegal use of private packages. Secondly, developers should strive for stability when designing the signatures of API methods because changes are almost always binary incompatible. Unfortunately, we suspect that most developers are only familiar with rules of source compatibility, and not aware of the subtle differences between binary and source compatibility. Thirdly, programs should not

rely on private JRE packages (`sun.*` and similar) because these packages might not be available on alternative Java platforms such as Android. Finally, public constants should be used only for values that are never to be changed, e.g. physical constants, otherwise inlining can cause applications to rely on obsolete values.

We believe partial upgrades are a trend that is only going to increase, supported by new technologies like OSGi or the upcoming integration of the project jigsaw into Java [28]. Apart from the above recommendations, the findings of our study therefore provide some motivation for more research towards improved consistency between the Java compiler and the Java virtual machine.

Some relatively minor changes to standard development tools and the Java language could be very effective, for example a compiler annotation to prevent constant inlining or features to restrict the access to packages. Without explicit language support, developers have to resort to less elegant solutions like preventing inlining by using wrapper types instead of the respective primitive types. We found that this defensive programming technique is used in several open source projects, including *ivatagroupware-0.11.3* and *velocity-1.6.4*. Auto-unboxing makes this transparent to other applications. However, there is no guarantee that future versions of the compiler will not inline these constants, and the intention why these types are used should be communicated to developers to ensure that they are used correctly and not accidentally converted back to primitive types.

ACKNOWLEDGMENTS

The work was partially supported by European Regional Development Fund (ERDF), project "NTIS - New Technologies for the Information Society", European Centre of Excellence, CZ.1.05/1.1.00/02.0090.

REFERENCES

[1] Miles Barr and Susan Eisenbach. Safe upgrading without restarting. In *Proceedings ICSM'03*, pages 129–137, 2003.

[2] Jaroslav Bauml and Premek Brada. Automated versioning in OSGi: A mechanism for component software consistency guarantee. In *Proceedings 35th EUROMICRO'09*, pages 428–435. IEEE Computer Society, 2009.

[3] Premek Brada. Enhanced type-based component compatibility using deployment context information. *Electronic Notes on Theoretical Computer Science*, 279(2):17–31, December 2011. Proceedings FESCA'11.

[4] Premysl Brada and Lukas Valenta. Practical verification of component substitutability using subtype relation. In *Proceedings EUROMICRO'06*, pages 38–45. IEEE Computer Society Press, 2006.

[5] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30, 2002.

[6] John Corwin, David F. Bacon, David Grove, and Chet Murthy. Mj: a rational module system for java and its applications. *SIGPLAN Not.*, 38(11):241–254, October 2003.

[7] Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings FSE'12*, page 55. ACM, 2012.

[8] Brad J Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[9] Ilie Şavga and Michael Rudolf. Refactoring-based support for binary compatibility in evolving frameworks. In *Proceedings GPCE '07*, pages 175–184, New York, NY, USA, 2007. ACM.

[10] Jim des Rivières. Evolving Java-based APIs. http://wiki.eclipse.org/Evolving_Java-based_APIs. [Accessed: Aug. 28, 2013], 2007.

[11] Jens Dietrich and Lucia Stewart. Component Contracts in Eclipse-A Case Study. In *Proceedings CBSE'10*, pages 150–165. Springer, 2010.

[12] Danny Dig and Ralph Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):83–107, 2006.

[13] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. ReBA: refactoring-aware binary adaptation of evolving libraries. In *Proceedings ICSE '08*, pages 441–450, New York, NY, USA, 2008. ACM.

[14] Mikhail Dmitriev. Language-specific make technology for the Java programming language. In *Proceedings OOPSLA '02*, pages 373–385, New York, NY, USA, 2002. ACM.

[15] Sophia Drossopoulou, David Wragg, and Susan Eisenbach. What is Java binary compatibility? In *ACM SIGPLAN Notices*, volume 33, pages 341–361. ACM, 1998.

[16] Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *Proceedings OOPSLA '95*, pages 426–438, New York, NY, USA, 1995. ACM.

[17] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java ™Language Specification 7th Edition*. Oracle, Inc., California, USA, 2012.

[18] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. On the extent and nature of software reuse in open source java projects. In Klaus Schmid, editor, *Top Productivity through Software Reuse*, volume 6727 of *LNCS*, pages 207–222. Springer Berlin Heidelberg, 2011.

[19] JBoss, Inc. JBoss Project Versioning. https://community.jboss.org/wiki/JBossProjectVersioning. [Accessed: Sept. 2, 2013], 2012.

[20] Kamil Jezek, Lukas Holy, and Premek Brada. Supplying compiler's static compatibility checks by the analysis of third-party libraries. In *Proceedings CSMR'13*, pages 375–378. IEEE Computer Society, 2013.

[21] Ralph Keller and Urs Hölzle. Binary Component Adaptation. In *Proceedings ECOOP '98*, volume 1445 of *Lecture Notes in Computer Science*, pages 307–329. Springer, 1998.

[22] Ralf Lämmel, Ekaterina Pek, and Jürgen Starek. Large-scale, AST-based API-usage analysis of open-source Java projects. In *Proceedings SAC'11*, 2011.

[23] M. M. Lehman and L. A. Belady, editors. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[24] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™Virtual Machine Specification - Java™SE 7 Edition*. Oracle, Inc., 2012.

[25] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[26] Tom Mens, Juan Fernández-Ramil, and Sylvain Degrandsart. The Evolution of Eclipse. In *Proceedings ICSM'08*, pages 386–395, October 2008.

[27] Oracle, Inc. Java ™Product Versioning. http://docs.oracle.com/javase/7/docs/technotes/guides/versioning/spec/versioning2.html. [Accessed: Aug. 28, 2013], 2013.

[28] Oracle, Inc. Project Jigsaw. http://openjdk.java.net/projects/jigsaw/. [Accessed: Aug. 28, 2013], 2013.

[29] OSGi Alliance. Semantic Versioning Technical Whitepaper Revision 1.0. http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf. [Accessed: Aug. 28, 2013], 2010.

[30] OSGi Alliance. OSGi Service Platform Release 4.3. http://www.osgi.org/Release4/Download. [Accessed: Aug. 28, 2013], 2012.

[31] Steven Raemaekers, Arie van Deursen, and Joost Visser. Exploring risks in the usage of third-party libraries. *Software Improvement Group, Tech. Rep*, 2011.

[32] Widura Schwittek and Stefan Eicker. A study on third party component reuse in Java enterprise open source software. In *Proceedings CBSE '13*, pages 75–80, New York, NY, USA, 2013. ACM.

[33] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies. In *Proceedings APSEC'2010*, pages 336–345. IEEE, 2010.

[34] Terry Winograd. Beyond programming languages. *Commun. ACM*, 22(7):391–401, July 1979.