Information and Software Technology 65 (2015) 129–146

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

How Java APIs break – An empirical study

Kamil Jezek [a], Jens Dietrich [b,*], Premek Brada [a]

[a] NTIS – New Technologies for the Information Society, European Centre of Excellence, Faculty of Applied Sciences, University of West Bohemia, Univerzitni 8, 306 14 Pilsen, Czech Republic
[b] School of Engineering and Advanced Technology, Massey University, Palmerston North, New Zealand

ARTICLE INFO

Article history:
Received 22 June 2014
Received in revised form 12 February 2015
Accepted 26 February 2015
Available online 6 March 2015

Keywords:
Binary compatibility
API evolution
Backward compatibility
Byte-code
Java

ABSTRACT

Context: 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.
Objective: We have studied the extent of the problem in real world programs. We were interested in two aspects: the compatibility of API changes as libraries evolve, and the impact this has on programs using these libraries.
Method: This study is based on the qualitas corpus version 20120401. A data set consisting of 109 Java open-source programs and 564 program versions was used from this corpus. We have investigated two types of library dependencies: explicit dependencies to embedded libraries, and dependencies defined by symbolic references in Maven build files that are resolved at build time. We have used JaCC for API analysis, this tool is based on the popular ASM byte code analysis library.
Results: We found that for most of the programs we investigated, APIs are unstable as incompatible changes are common. Surprisingly, there are more compatibility problems in projects that use automated dependency resolution. However, we found only a few cases where this has an actual impact on other programs using such an API.
Conclusion: It is concluded that API instability is common and causes problems for programs using these APIs. Therefore, better tools and methods are needed to safeguard library evolution.

1. Introduction

Re-use has long been seen as an important approach to reduce the cost and increase the quality of software systems [46,11]. One of the re-use success stories is the use of libraries (jar files) in Java programs [23,43]. 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. 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 high availability (“24/7 systems”) and software 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

* Corresponding author.
E-mail addresses: kjezek@kiv.zcu.cz (K. Jezek), j.b.dietrich@massey.ac.nz (J. Dietrich), brada@kiv.zcu.cz (P. Brada).

http://dx.doi.org/10.1016/j.infsof.2015.02.014
0950-5849/

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 [30, Chapter 5.3.2],[22, Chapter 12.7]. The most successful implementation of this idea to date is OSGi [38], 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 (Oracle WebLogic, IBM 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 between these sets of rules 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, auto-(un) boxing 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" [35].

A commonly used solution is to add another layer of constraints to restrict linking. For instance, OSGi-based systems should use semantic versioning [37]. 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 [2]: (a) Many code changes are very subtle (as discussed below) and it cannot 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 (c) Creators and users of a library must have a common understanding of the versioning scheme they use. Therefore, any approach that relies on manually assigned versions is inherently error-prone.

The objective of this paper is to investigate how Java APIs evolve and to study to what degree compatibility issues occur in practice. In particular, we are interested in the following research questions:

- RQ1 How frequent are API-breaking changes when programs evolve?
- RQ2 How do incompatible changes affect client programs – at build time during compilation or at link time?
- RQ3 How many actual programs are affected by API-breaking changes?
- RQ4 Is there a pattern correlating API-breaking changes and versioning schemes?

This paper is organised as follows. We review related work in Section 2, discuss separate types of compatibility in Section 3 and classify the evolution problems we want to investigate in Section 4. Section 5 describes the methodology used to set up and execute the experiments. The results of these experiments are reported in Section 6, followed by a discussion of threats to validity in Section 7 and the conclusion in Section 8.

2. Related work

The notion of binary compatibility goes back to Forman et al. [21], 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 [22, Chapter 13]. Drossopoulou et al. [20] have proposed a formal model of binary compatibility. A comprehensive catalogue of binary compatibility problems in Java has been provided by des Rivières [13]. The problems we have investigated here are a subset of this catalogue.

The more general notion of compatibility between collaborating components has been studied by Beugnard et al. [4]. The authors pointed out that there are several types of contracts collaborating software components have to comply with in order to collaborate successfully. The focus of this study is on syntactic contracts that can be checked by investigating the type system. Belguidoum et al. suggested the notions of horizontal and vertical compatibility [3]. Our work is based on this conceptual framework, as discussed in Section 3.

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 [19] has proposed to use binary compatibility checks in 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) [27] is based on the idea to manipulate class definitions at runtime to overcome certain binary compatibility problems. Dig et al. [18] and Savga and Rudolf [12] have proposed a refactoring-based solution to generate a compatibility layer that ensures binary compatibility when referenced libraries evolve. Corwin [9] 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 [17] 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*" [17]).

Mens et al. [34] 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 [29]. 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 [10]. 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 [16]. 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 [44] in our study. This data set has been widely used in empirical studies, including several studies on how APIs and libraries are used [40,23,43]. Lämmel et al. [28] 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.

Raemakers, van Deursen and Visser have recently conducted a number of studies that are closely related to the work presented here. In [40], they have studied the level of reuse amongst the programs and libraries in the Qualitas Corpus, and suggested metrics to quantify the level of reuse. In [41], the same authors studied the stability of one particular library (`apache-commons`), and how it affected other programs using it. In this study, we conduct a similar experiment for an larger set of libraries. The notion of stability used in [41] is defined with respect to a simple coarse notion of (method) change and removal, whereas we use a more fine-grained classification of multiple change catagories. In [42], the same authors studied the relationship between semantic versioning and API stability in programs from the Maven repository, and found that developers do not adhere to the principles of semantic versioning in practise, and that breaking changes are common in minor and patch releases. This is similar to results we present in this paper. The main difference is that we use different tooling that supports a more precise analysis as we do not only consider the rules of binary compatibility, but also include more subtle compatibility issues related to source and certain types of behavioural compatibility. In [42], the authors have used the Clirr tool which only checks for binary compatibility, and therefore may have produced only a coarse under-approximation of compatibility issues. Furthermore, our work includes the analysis of how the breaking changes actually affect clients of the changed programs.

This paper is an extension of our previous work [14]. In particular, we added the analysis of dependencies defined by symbolic references in Maven build files to the study, whereas in [14] we had only considered dependencies on libraries distributed with programs. This paper also includes a broader discussion of several concepts of compatibility (see Section 3).

3. On the notion of compatibility

Evolution problems arise when a library providing functionality through an API to a program is replaced by a different library. The notion of compatibility defines when this replacement is safe, and when it is not. We consider a library A to be different from library B if there is *any* kind of change. This includes not only changes to the actual code representing the functionality of the library, but also changes to properties like name, version number, and other meta- and configuration data, including licenses and deployment descriptors. In particular, a newer version of a library is always considered as being different from an older versions. The question whether replacing a library is safe is also called the substitutability problem [7]. If we focus only on the provided functionality (leading to the notion of API compatibility), then the theoretical solution is similar to how substitutability is approached in subtyping [31]: substitution is safe if the preconditions of invoked services are not strengthened, and the postconditions are not weakened.

If a program uses a library, both engage in a contractual provider-consumer relationship. A simple example for such contracts is the use of interfaces in Java – the client is in a consumer role requesting a class implementing this interface from a supplier. This view is promoted by classical software design principles, such as the interface segregation principle [33] and the dependency inversion principle [32]. However, in practice contracts are often more complicated and also include aspects such as behavioural constraints (which cannot be expressed using programming language type systems), quality of service attributes and even aspects such as license compatibility [4]. We refer to this notion of contractual compatibility as *horizontal compatibility*, as suggested in [3].

On the other hand, it is also common to consider compatibility in the context of upgrading or replacing. In this case a component B is compatible with A if A can be replaced by B without breaking its contracts. We refer to this as *vertical compatibility* [3]. This includes the notion of backward compatibility which means that if A (as provider) is horizontally compatible with C (as consumer) and B is vertically compatible with A, then it can be inferred that B is also horizontally compatible with C. The relationship between vertical and horizontal compatibility is depicted in Fig. 1.

While such a simple model is appealing, practical scenarios are often more complicated and several other parameters come into play. Firstly, full vertical compatibility is almost impossible to achieve. Even minor changes to a provider can break a consumer that uses reflection (perhaps even including byte code analysis of the provider), or has logic that depends on timing characteristics of the provider, such as timeouts. Therefore, in practice we often only consider *contextual vertical compatibility* where we make additional assumptions about the consumers and the application context we allow. For instance, we may assume that clients do not use reflection to reason about (method) signatures, or that certain methods are only invoked, but not overridden. This will be discussed further in Section 4.

Secondly, the practical interpretation of horizontal compatibility, and therefore indirectly also vertical compatibility, depends on the actual technologies used to check it. An example for this is binary compatibility in Java. This is defined with respect to linking in the Java Language Specification [22], and checked by the JVM when classes are loaded. On the other hand, the Java compiler applies a different set of rules – source compatibility – to enforce horizontal compatibility at compile time. It turns out that neither binary compatibility implies source compatibility, nor vice versa, making the notion of compatibility relative with respect to the tool used for verification. Examples for situations where source and binary compatibility differ will be discussed below.

Both the linker and the compiler only check for signature compatibility. They do not investigate the actual semantics of methods, but only check their parameter and return types. In this study we do the same – we only investigate compatibility changes that result from API changes. We therefore have to assume that our findings are only the tip of the iceberg, and that there are many

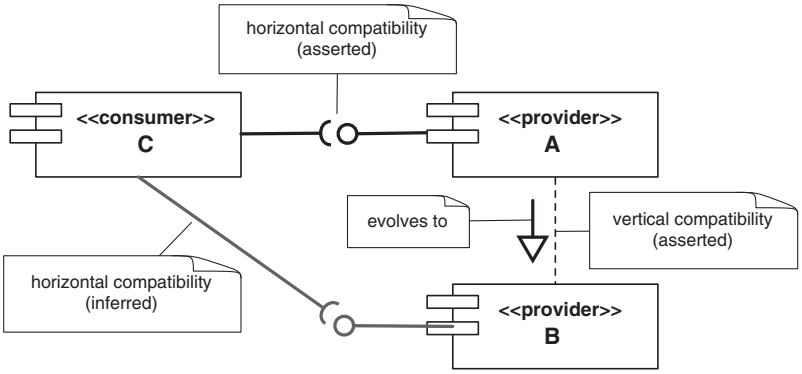

Fig. 1. Horizontal vs. vertical compatibility.

more "semantic" problems. Often, API changes indicate semantic changes, but we believe that there are many situations where the actual program semantics (in particular the logic represented by method bodies) is changed without changing the API (method signatures). After all, many object-oriented design principles promote API stability. There are rare cases when method signatures are changed in order to explicitly signal semantic change. An example is the change of collection types in method signatures from `Collection` to `List` or `Set`. The purpose here is often not to provide additional functionality through an extended API, but to communicate certain semantic properties of the respective collection, such as the handling of duplicates, the retention of the insert order or even the expected computational complexity of element lookups.

4. Evolution problems

There are several types of compatibility problems between evolving libraries and the programs using them. *Source incompatible* changes result in compilation errors as defined in [22] when a program is recompiled with the changed version of a library it depends on. *Binary incompatible* changes result in errors or unexpected behaviour when an existing program is executed with a changed version of a library. The (vertical version) of binary compatibility is defined in [22, Chapter 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.

We therefore use an extended definition of binary compatibility that also includes issues currently not checked by the Java linker. However, these issues can be checked by means of static analysis. An example is the handling of generic parameter types. The Java linker only checks horizontal binary compatibility. Since the Java compiler applies erasure and generic parameter types are removed from method references, the linker can only use the erasure when resolving method references. On the other hand, when investigating the vertical compatibility between two versions of a library providing an API, it is possible to compare the type parameters by analysing the signatures of the methods in both versions. This makes it possible to detect cases of vertical incompatibility that are not covered by the definition of binary compatibility in the Java Language Specification. In particular, changing generic type parameters does by definition not result in linkage errors, but may result in compilation errors and/or cause runtime exceptions. In particular, cast statements inserted by the compiler into consumer code may fail with `ClassCastExceptions` if type parameters are changed. This is further discussed below in Section 4.4.

A comprehensive catalogue of evolution problems is presented by des Rivières [13]. 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. In the text, a *program* denotes a non-empty set of binary components that form a single software product, a *library* denotes one such component, and a *change* denotes the difference in the implementation between two versions of the same item (library or program). For any pair of program versions, a non-zero number of changes of a certain kind is counted as only one change occurrence. For instance, if multiple methods are removed from classes in a program, only one change in the removed method category is recorded.

4.1. 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 is not referenced by a program, then the respective change is contextually compatible [6] – 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.

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

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.

4.2. 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 Section 4.1. Some of the categories in this group are special cases of a more general category defined in Section 4.1. The remainder of this section develops a list of these change categories, starting with the example in Listing 1.

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 [31]) – to honour the contract between the client program and the (old) API method, preconditions must not be strengthened and postconditions must not be weakened [13]. 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 upgraded library. The JVM tries to resolve the method reference with a method "with the name and descriptor specified by the method reference" [30, Chapter 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. This means that these errors 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 specialised or generalised parameter types. 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* [5], 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 [22, Chapter 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 [22, Chapter 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

```
1  // library version 1
2  public class Foo {
3    public static java.util.Collection foo() {return null;}
4  }
5  // library version 2
6  public class Foo {
7    public static java.util.List foo() {return null;}
8  }
9  // client program using the library
10 public class Main {
11   public static void main(String[] args) {
12         java.util.Collection list = Foo.foo();
13         System.out.println(list);
14   }
15 }
```

Listing 1. A source compatible but binary incompatible evolution.

auto unboxing, respectively [22, Chapter 5.1]. There are two permitted combinations of conversions and boxing/unboxing [22, Chapter 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 as the compiler applies boxing and unboxing, respectively [22, Chapter 5.1]. 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.

Java uses a predefined set of byte-code instructions to access and invoke fields and methods [30, Chapter 2.11]. While the compiler generates correct instructions for a particular context, incorrect instructions may appear when the byte-code is updated without recompilation.

An example when this happens is the invocation of static and non-static methods. The byte code instruction `invokestatic` is used for static methods, while `invokevirtual`, `invokeinterface`, `invokespecial` or `invokedynamic` are used to invoke non-static ones. These different instructions correspond to different dispatch rules used by the JVM. On the other hand, static methods can be invoked from non-static contexts in source code. This causes problems when the original code invoked a non-static method, which was then changed to a static method in a library that is compiled separately. In this case, the method is referenced using the wrong instruction (`invokevirtual`) and the linker reports an error (more precisely, an instance of `IncompatibleClassChangeError`). This problem can be easily solved by recompiling the client, as this will create the correct instruction (`invokestatic`).

The situation is similar for fields. Static fields can be also accessed from non-static contexts. The byte-code instructions `putstatic` and `getstatic` are used to access static fields, while `getfield` and `putfield` are used to access non-static ones. When a field originally set to non static is changed to static, clients must be recompiled to ensure that they use the correct instructions.

The last case involves interfaces and classes. Interface methods are invoked using a special `invokeinterface` instruction. When an interface is changed to a class or vice versa, clients suddenly use the wrong instruction. To solve this, they must be recompiled.

This leads to the definition of three additional compatibility categories for methods, fields and interfaces, respectively:

- M.M1 A non static method is changed to the static one.
- F7 A non static field is changed to the static one.
- C3 A class is changed to an interface or vice versa.

4.3. 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 [22, Chapter 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.

4.4. 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>`.

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.

```
1  // library version 1
2  public class Foo {
3         public static java.util.List<String> foo() {
4                return new java.util.ArrayList<String>();
5         }
6  }
7  // library version 2
8  public class Foo {
9         public static java.util.List<Integer> foo() {
10               return new java.util.ArrayList<Integer>();
11        }
12 }
13 // client program using the library
14 public class Main {
15        public static void main(String[] args) {
16               java.util.List<String> list = Foo.foo();
17               System.out.println(list.size());
18        }
19 }
```

Listing 2. A binary compatible but source incompatible evolution.

5. Methodology

To find out how frequently the evolution problems described above occur, and to measure the impact these changes have on *actual* library clients, we performed a set of experiments on real-world programs. In this section, we briefly describe the setup of these experiments. The results of these experiments are described in the next section.

In this study, we investigate *programs* – software systems that provide a certain functionality in order to serve a well-defined purpose. We sometimes refer to programs as projects if we want to emphasise the aspect that a program has a certain internal structure and organisation that may effect its function and evolution.

5.1. Data set

We have studied the API evolution of programs contained in the qualitas corpus [44]. 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 [39,23,43].

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.

5.2. 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 program 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 into account project-specific versioning schemes like the scheme used by the JBoss project [24] (Alpha, Beta suffixes for test versions and GA for releases – General Availability).

5.3. Cross-referencing programs – embedded dependencies

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 given program. 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 program) and *third party libraries* (used by the program). This was done by recursively searching the respective program version folders for jar files, and classifying files with names containing the program name as program libraries, and all other jar files as third party libraries.
2. In a second step, third party libraries used in programs were matched to program libraries in the program 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 programs using them, for instance, 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 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.

Using this process, we found that 212 library versions are actually being used by any version of other programs in the corpus. Interestingly, the number of libraries included in programs as .jar files is 910. This means that only 23% of libraries are actually used by other programs in the corpus.

Fig. 2 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

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

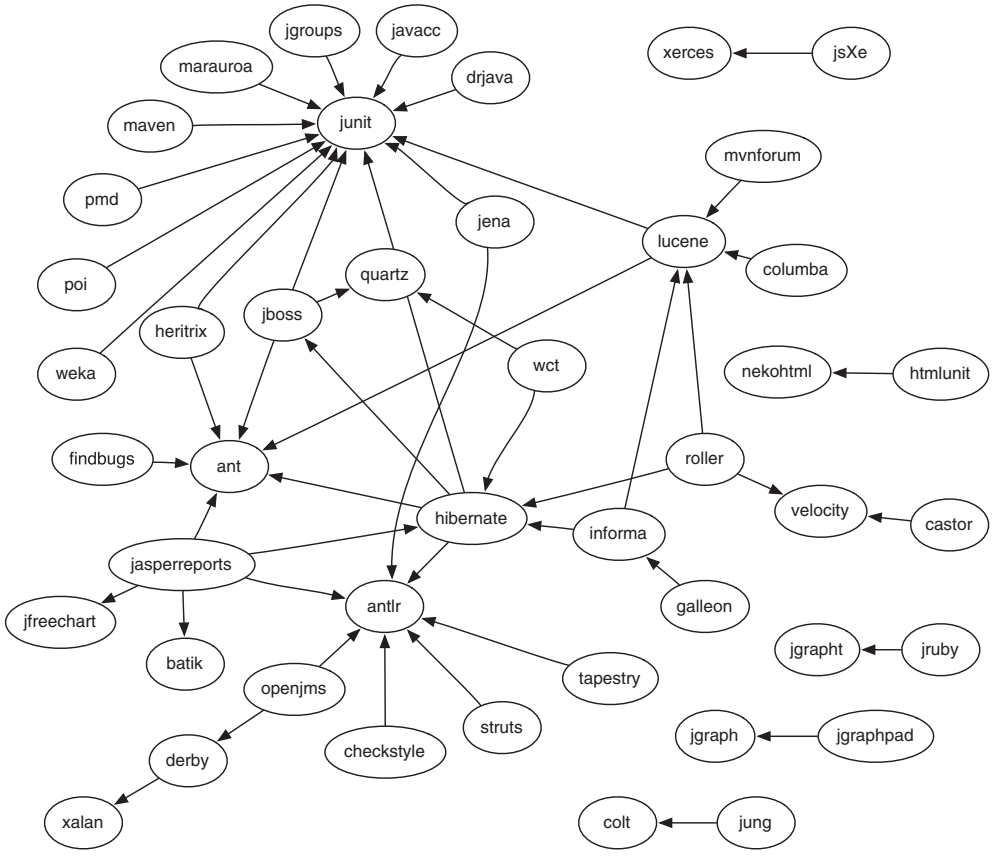

Fig. 2. Cross-project dependencies between programs in the qualitas corpus defined by libraries included in projects.

Table 1
Popular programme 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

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 1 shows selected program versions and the number of other program versions in the corpus which use any of the libraries they provide.

5.4. Cross-referencing programs – symbolic references

In many cases, programs do not include their dependencies (third party libraries) but reference them using symbolic references that are resolved at build time against library repositories. Many projects used the popular Maven build tool[3] and its associated repository for this purpose.

This observation lead us to an extension of the experiments where we considered additional dependencies defined by Maven. All programs that have at least one `pom.xml` file (the Maven build script) were considered as Maven projects for this purpose. This resulted in the detection of 30 Maven projects with 73 versions. Often, Maven projects are divided into *modules* which have separate `pom.xml` files. This is also the case for corpus programs; in the qualitas corpus we detected 1894 modules of the 73 program versions.

In Maven projects, library dependencies are declared in the `pom.xml` files. These dependencies are symbolic references to resources stored in Maven repositories. In most cases, the central repository[4] is used. We used the JCabi-aether[5] tool to resolve symbolic references and obtain the respective libraries (byte code) from the central repository. Note that JCabiaether has the capability to recursively resolve dependencies, i.e., it computes the transitive closure of the dependency set.

The number of 3rd party libraries referenced via Maven is very high: the 73 program versions directly or indirectly reference a total of 5159 libraries. While there certainly is an intersection of this Maven-based set of library versions with the set contained in the qualitas corpus proper, identifying this intersection would be a very tedious process requiring pair-wise binary comparison of all libraries; as discussed in subSection 5.3. We therefore performed the experiments separately on both sets. The dependencies

[3] http://maven.apache.org/.
[4] http://search.maven.org/.
[5] http://aether.jcabi.com/.

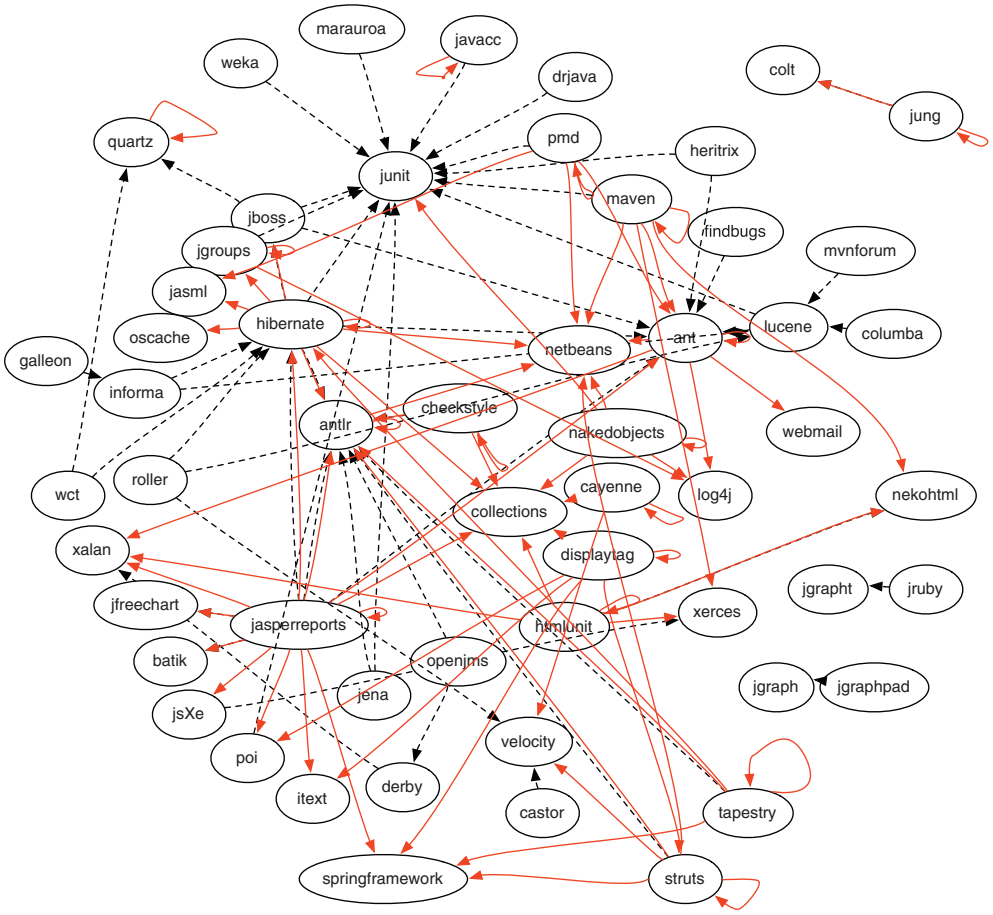

Fig. 3. Cross-project dependencies between programs in the qualitas corpus defined by libraries included in projects and symbolic references defined in maven build scripts (solid red lines). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

were again analysed by the DependencyFinder tool in order to check whether a linked library is actually being used. It turns out that only 3988 libraries are actually used.

As discussed before, it is difficult to assess whether a Maven module belongs to a particular project as Maven module names are not necessarily the same as project names. To mitigate this problem, we gathered the Maven's artefact IDs of all modules defined by programs in the corpus and matched dependencies using these IDs. For non-Maven projects we tried to guess the match by comparing JAR file names. This is a viable approach as it is common practise in Maven that a final artefact is named using the respective project name. However, this yields some false positives. For instance, the (corpus) project `springframework` defines several Maven modules `spring-bean`, `spring-aop`, etc. For this reason, we did some manual corrections to improve precision. The results of this process are shown in Fig. 3. It depicts the aggregated project dependency graph now also including Maven dependencies (highlighted by solid red lines). Note that many more dependencies were found in Maven projects (3988) but this graph shows only links between those programs contained in the qualitas corpus – 122 dependencies in total. Vertices with self edges represent dependencies between multiple modules defined within the same project.

5.5. Code compatibility analysis

We used mainly byte code analysis tools, in particular ASM [8] and JaCC [26] version 1.0.2, to investigate the compatibility between libraries and programs. The latter tool is specifically developed for API compatibility analysis, internally using ASM but providing 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 ad hoc scripts for this purpose that use simple regular expression matching.

6. Results

In this section, we present the results of the various experiments conducted to investigate to which extent the change categories discussed in Section 4 occur in practice.[6] These experiments are based on the techniques described above and use the qualitas corpus data set plus the extended data obtained by resolving Maven dependencies, as explained in subSection 5.4.

6.1. Program distribution completeness

In a first set of experiments we have checked the corpus programs for distribution completeness. In total, 564 program versions were analysed. We investigated whether all classes referenced in the given program version are defined either in the program itself or in one of the libraries distributed with it.

We detected several cases of incomplete distributions, falling into two categories. Firstly, there are cases where referenced

[6] Raw data are available at: http://relisa-dev.kiv.zcu.cz/data/experiments/api-evolution-problems-2014-06/.

classes are defined in third-party libraries that are not included in the program distribution. We found 480 program versions where this is the case for at least one class. This means that almost all program versions investigated were affected by this problem. Even when Maven-defined dependencies were added, this number did not change. The practical consequence is that the deployment of such programs needs to be performed carefully, adding the not included libraries manually.

For instance, there are programs with references to *junit* types where the *junit* library is not part of the program (24 cases in total, 18 when the library is obtained by resolving the Maven references). 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 a third party library. In Maven projects, *junit* is often missing because it is part of the "test" scope. We only checked the dependencies defined in the "runtime" scope.

Secondly, we found a significant number of project versions (82) referencing classes in `com.sun.*` packages (973 such references in total). We also found 34 project versions 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.

6.2. Horizontal compatibility

Next we investigated the horizontal compatibility between programs and libraries they use. Here we looked for cases where programs use some classes provided by certain libraries, and these classes are present in the respective libraries but do not provide the API used by the program.

We detected only one such case in a library dependency not defined by Maven. This was a case belonging to the M2 category that occurs in `hibernate-3.3.2-ga`. This program references the class `org.jboss.cache.Fqn`. This class is instantiated using constructors with the following signatures (package identifiers are omitted for brevity): `Fqn(Fqn,List)` and `Fqn(Fqn,Object)`. These constructors were not found in the library providing this class. However, manual investigation of the problem showed that `Fqn` is provided by *two* libraries available within the program: `jboss-cache-1.4.1.GA.jar` and `jbosscache-core-3.1.0.GA.jar`. Interestingly, only the second library provides the class with the correct constructors. This implies that the consistency of the program depends on which library is actually being used. It is not clear why this program contains two libraries which each provides a version of the same class.

This points at a wider problem of Java. The Java classpath may contain several libraries each defining a class with the same fully qualified name. At runtime, the classloading mechanism resolves these redundancies by allowing only the first encountered class version to be loaded per class loader. This makes it difficult to foresee which class will actually be used, resulting in unpredictable program behaviour. This problem is sometime called "JAR hell" or "classpath hell" by developers in reference to the "DLL hell" problem that has plagued Windows applications for a long time.

We also investigated the 73 project versions with Maven-defined dependencies for horizontal compatibility between the projects and the libraries they use. Every project in this group has at least one missing class (C1 category), 31 projects (about half in total) referenced incompatible classes (C2) or missing methods (M1). Finally, 7 projects invoked incompatible methods (M2). Note that all of these problems are binary as well as source incompatible.

A particularly interesting example is `struts-2.2.1`. This project version invokes the method `parse(URL)` of `org.apache.commons.digester.Digester`. This class is provided by the project `commons-digester` and the method has been added in version 1.7. However, `struts-2.2.1` references *three* different versions of the digester: 1.6, 1.8 and 2.0. These are indirect references defined by referenced sub-modules. As version 1.6 of `Digester` does not provide a required `parse` method, it is incompatible with `struts-2.2.1`.

In the 73 project versions with Maven-defined dependencies, we found six additional horizontal compatibility problems that can all be solved by recompilation (binary incompatible, but source compatible). All of these problems appear in versions of the `hibernate` project and belong to M.P1 category.

In particular, the following two M.P1 problems were repeated in all cases (the results are shown in a form <invoked> x <provided> type, the number in brackets following the method name is the number of parameters):

```
Class: org.jboss.cache.Fqn
  Constructor: Fqn(2)
    Argument: ...
    Argument: java.lang.Object[] x java.lang.Object

Class: org.jboss.cache.Node
  Method: getChild(1)
    Argument: org.jboss.cache.Fqn x
   java.lang.Object
```

In these examples, methods or constructors provided by the `jboss` cache are invoked. This is similar to the example discussed above. Manual analysis reveals that these problems are also caused by inconsistencies between multiple co-existing cache libraries. In particular, some versions of the hibernate project (for instance, 3.3.0-ga) contain `org.hibernate:hibernate-jbosscache:jar:3.3.0.GA` which references `jboss:jboss-cache:jar:1.4.1.GA`. This means that two libraries are used which both provide (inconsistent versions of) the same type. Since the libraries have different identifiers, Maven treats them as independent and includes them both in the classpath.

These results indicate that the use of tools like Maven providing automated dependency management does not prevent compatibility errors. On the contrary, the results suggest that the use of Maven can lead to duplicated and inconsistent dependencies, and can therefore even increase the probability of compatibility errors occurring. A more detailed discussion on this subject is provided in [25].

6.3. Vertical compatibility – binary and source incompatible changes

We investigated 455 version pairs representing "atomic" upgrades, this is, upgrades to an adjacent version of the same program according to the version order described in Section 5. The goal was to detect changes of program libraries which could potentially cause incompatibilities in applications using this project that would require refactoring.

We found that 375 version pairs have incompatibilities and only 74 version pairs are compatible. In other words, 80% of upgrades are not compatible and the number of incompatible upgrades is about 5 times higher than compatible upgrades. Details are shown in Table 2 for some frequently used programs. It provides the number of versions of the respective program where at least one evolution problem from any of the categories defined above was detected. Note that these results are not

Table 2
Distribution of binary and source incompatible API changes by category.

Category	junit	hibernate	lucene	ant	antlr	All	Mean	Variance
C1	13	76	23	15	11	284	54.48	48701.10
C2	22	94	25	19	18	362	263.11	315014.79
F1	9	60	21	13	13	252	100.53	357693.69
F2	4	26	16	5	9	147	14.53	13439.10
M1	11	85	23	16	15	289	97.73	146267.59
M2	20	91	24	15	15	316	4611.87	792800738.43
MOD	19	89	21	14	13	309	4509.53	771627520.15

normalised, even a single incompatibility issue is enough to classify a version upgrade as incompatible.

This table also shows the mean values of occurrences of the respective problems across all programs investigated for each category and the respective variance values in last two columns. The mean value indicate that the most common problems belong to the M2 and MOD categories. The high values indicate that these changes are very common in some programs.

We also investigated how many programs in the corpus had completely stable APIs. That is, we were interested in programs with no adjacent version pairs exhibiting the incompatibility problems defined in Section 4.1. We only included programs with four or more versions because API stability in programs with only a few versions might occur accidentally and would therefore not be very meaningful. Therefore, 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*. Other programs are riddled with incompatible changes as shown in Table 3. The table shows both the total number of versions of the respective program, and the number of versions that are compatible with the previous version. The last two columns of the table depict the mean and variance computed from all changes in all categories between program versions.

The mean values indicate that the most unstable API changes occur in `argouml`, followed by `hibernate`, `jhotdraw` and `freemind`. Other programs had a considerably smaller amount of incompatible changes. The high value of variance shows that amount of changes differ widely between program versions.

The results described so far allow us answer **RQ1: How frequent are API-breaking changes when programs evolve**? API-breaking changes are very common as 80% of version updates were detected as incompatible.

Moreover, the high variance indicate that the number of incompatibilities changes widely among program updates.

6.4. Vertical compatibility – binary incompatible but source compatible changes

In this experiment, we looked for incompatible changes that can be addressed by recompilation as defined in Section 4.2. These changes are relatively infrequent compared to the change categories defined in section 4.1. Table 4 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 within the corpus. This was not the case for any of these methods, and we therefore considered these methods as used-only as defined in Section 4.2.

The results presented in this and the previous section provide the data to answer **RQ2: How do incompatible changes affect client programs – at build time during compilation or at link time?** The obtained numbers show that most of the incompatible changes are caused by source incompatibilities. Therefore, clients will be affected early at build time when compilation fails. Although problems causing only binary incompatibilities were relatively rare, some cases from almost all categories (except F3-F5, M.R4) were detected. This shows that programs are also prone to run- and link-time only failures.

This section concludes with some examples of incompatibilities found. Each listing starts with the change category as defined in Section 4.1, followed by the two versions of the library that were compared, the class (type) 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>`.

Table 3
Number of compatible version changes that are API stable for some popular libraries.

Programme	Version changes	Compatible version changes	Mean	Variance
ant	20	1	22.09	552.25
antlr	19	1	30.45	3802.78
argouml	15	1	1901.31	7773051.68
colt	4	0	21.41	417.33
freecol	27	27	0.00	0.00
freemind	15	1	160.33	91650.26
hibernate	99	2	243.54	1325690.56
jgraph	38	14	3.52	56.11
jhotdraw	5	0	198.62	30923.64
jmeter	19	19	0.00	0.00
jung	22	0	73.40	32975.11
junit	22	0	20.99	5374.19
lucene	27	1	44.15	6650.04
weka	54	2	66.04	40742.49

Table 4
Distribution of binary incompatible but source compatible API changes by category.

Category	junit	hibernate	lucene	ant	antlr	All	Mean	Variance
M.P1	2	22	7	1	3	81	2.83	208.85
M.P2	0	0	2	0	0	16	0.19	4.36
M.P3	0	3	0	0	0	4	0.02	0.09
M.P4	0	0	0	0	0	1	0.01	0.05
M.R1	1	18	4	0	0	54	2.58	650.72
M.R2	0	0	1	0	0	4	0.08	2.41
M.R3	0	3	0	0	0	6	0.12	2.89
M.R4	0	0	0	0	0	0	0.00	0.00
M.M1	2	5	0	0	0	29	1.07	220.20
F3-F5	0	0	0	0	0	0	0.00	0.00
F6	0	7	5	1	1	22	0.19	1.62
F7	0	1	0	1	3	15	0.28	7.16
C3	0	11	1	1	0	34	0.12	0.24

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.SimpleExpression
  Argument:
    org.hibernate.criterion.DetachedCriteria
    -> 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
```

Example F6. poi-2.5.1 → poi-3.6.

```
Class: org.apache.poi.hssf.usermodel.
  HSSFErrorConstants
  Field: ERROR_NA
    Type: byte -> int
```

Example F7. pmd-3.3 → pmd-4.2.5

```
Class:
  net.sourceforge.pmd.renderers.EmacsRenderer
  Field: EOL
    Modifier: protected -> protected static final
```

Example M.M1. poi-2.5.1 → poi-3.6.

```
Class: org.apache.poi.ddf.EscherBSERecord
  Method: getBlipType(1)
    Modifier: public x public static
```

Example C3. hsqldb-1.8.0.4 → hsqldb-2.0.0

```
Class: org.hsqldb.TransactionManager
  Invocation: class -> interface
```

6.5. Impact on horizontal compatibility

In the next experiment, we checked whether any of the API-breaking changes we observed had an actual impact on other programs. In other terms, we studied whether vertical incompatibilities representing *potential problems* can turn into horizontal compatibilities representing *actual problems*.

For this purpose, we used the 212 dependencies between cross-referenced program versions (see Section 5.3) and the 3988 dependencies resulting from Maven projects (see Section 5.4). We then replaced the actually referenced program version by all available successive versions. How the successive versions (updates) were obtained differs for programs with embedded dependencies and Maven-defined (symbolic) dependencies.

For programs with embedded library dependencies, we obtained successive library versions by considering successive versions of the (corpus) programs defining them. This resulted in 5905 pairs consisting of a version of a program and the version of a library that is an upgrade of a library version the program uses.

For each project using Maven, we (recursively) downloaded the set of libraries this project depends on from the Maven repository. Therefore, the number of sets of libraries downloaded is the same as the number of program versions using Maven – 73. These library sets were then used in conjunction with all *older* program versions in order to simulate updates. This resulted in 414 additional pairs of program/library versions.

We ignored the dependencies that could not be resolved within the project – these are the dependencies described in Section 6.1.

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 *used* in P.

We detected numerous binary incompatible dependency changes from the categories defined in Section 4.1.

For programs with embedded libraries, we detected binary incompatibilities in the use of 66 library versions of only 6 libraries. In particular, 35 libraries exhibit compatibility problems due to removed classes (C1), 52 due to incompatible types (C2), in 25 cases referenced fields were removed (F1), 15 libraries contain incompatible field changes (F2), in 27 cases referenced methods were removed (M1), and 37 cases were caused by incompatible changes to methods (M2). We detected no issues caused by incompatible modifiers (MOD). These numbers are grouped together for all libraries for simplicity. In the next paragraphs this is discussed in more detail.

Comparing these numbers with the vertical (“potential”) incompatibilities figures in the last column of Table 2, gives us an “impact measure” in percentages – for example, 14.4% (52/362) of the potential problems due to incompatible type changes become real issues when partially upgrading the respective programs, and similarly 10.2% for field changes.

Although we found numerous compatibility problems, they affected only 12 client program versions. In particular, *jgroups-2.6.2* is incompatible with 13 *junit* versions, *hibernate-3.3.2-ga* with

Table 5
Ratio between errors and dependencies by dependency type.

Category	Maven-defined dependencies (415)	Embedded dependencies (5905)
C1	0.33	0.02
C2	0.92	0.04
F1	0.00	0.01
F2	0.00	0.01
M1	0.92	0.03
M2	0.54	0.02

10 *antlr* versions, *jasperreports-1.1.0* with two *jfreechart* versions, *columba-1.0* is incompatible with 22 *lucene* versions, *informa-0.6.5* with 22 *lucene* versions, *informa-0.7.0* with 18 *hibernate* versions and 22 *lucene* versions, *jasperreports-3.7.3* with 18 *hibernate* versions, *mvnforum-1.0* with 22 *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, and *jena-2.5.5* is incompatible with 13 *junit* 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.[7]

The same experiments were then performed for projects with Maven-defined dependencies. The number of compatibility issues obtained for these projects are 137 (C1), 381 (C2), 170 (F1), 381 (M1), 233 (M2), impacting 415 client program versions. In total 399 versions were detected for `hibernate`, 10 `anltr` versions, 4 `ant` versions and one `displaytag`. These values are not fully comparable with the values for non Maven projects because libraries were represented as sets representing the transitive closure with respect to the dependency relationship.

It is interesting to compare the relative numbers of problems for both types of dependencies (explicit and Maven-defined). Table 5 shows the ratio between the number of errors and the number of dependencies.

This table indicates that there are relatively more compatibility problems in projects using Maven. The reason may be that dependencies are resolved automatically by Maven and the user has only a limited possibility to customise the resolution. A consequence is that one wrong dependency may be propagated into other projects as dependencies are resolved recursively.

We also investigated the number of incompatible changes that can be resolved simply by recompiling the program with the new version of the respective library. We found only two such cases amongst programs using embedded libraries, but 199 cases amongst programs with Maven-defined dependencies.

The first case was detected in *jaspereports*. Its 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*.[8] 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
```

Although we found numerous problems in projects using Maven-defined dependencies (199 cases), these problems were caused by only a small number of binary incompatible changes in libraries used. This includes three changes in the `hibernate` project in the M.P1, M.R1 and C3 categories, respectively. In particular, 15 cases were in M.P1 caused by changes in the `jboss` cache. This is the same problem discussed before (see section 6.2).

We detected 14 cases in M.R1 caused by `createMappings` defined in `org.hibernate.cfg.Configuration` which was invoked with the return type `org.hibernate.cfg.Mappings` instead of using (its subtype) `org.hibernate.cfg.ExtendedMappings`. These problems only occurred when upgrading to beta versions of `hibernate-3.6.0` (`beta1` and `beta2`). For this reason, this problem would not have an impact on client programs as long as they avoided using (unstable) beta versions of libraries.

Finally, we detected 170 problems in C3 caused by changes in `org.hibernate.stat.SecondLevelCacheStatistics`. This type was changed from a class to an interface between the `3.3.0.SP1` and the `3.5.0.FINAL` version, though the new interface contains the same methods as the original class.

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.

It is again interesting to see that using Maven actually increased the number of incompatibilities found.

We conclude this section with an answer to **RQ3 How many actual programs are affected by API-breaking changes?** We did detect cases were source and binary incompatible changes broke actual client programs and forced these programs to be recompiled and/or refactored. On the other hand, the vast majority of incompatible changes detected in the previous sections has little impact on projects with explicitly managed dependencies. In particular, only 12 program version were affected. On the other hand, the problem increases once automated dependency resolution with Maven is used.

One reason for the relatively low number of actual problems is the Java language design. Java offers only very coarse features to implement encapsulation policies. In particular, every method that is used across different packages within a programme must be declared public, and becomes therefore part of the API. In many cases, this code is not intended for use from outside of the program. A good example for this are the classes in `sun.*` and `com.sun.*` packages in the Java platform itself. They are clearly not intended to be part of the API but are used like API classes

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

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

(see Section 6.1 for details) as they are public and provide useful functionality. This is a known limitation of the Java platform, and technologies like OSGi and JigSaw provide at least partial solutions by creating new units of encapsulations.

6.6. Vertical vs. horizontal compatibility

In the previous sections, we investigated vertical and horizontal compatibility separately. This raises the question what the relationship between these two categories is. In particular, which kinds of vertical compatibility problems are likely to have an impact on clients.

Fig. 4 shows vertical vs. horizontal incompatibilities by category. The bars represent the mean values of the number of vertical compatibility problems detected for the respective category. The values to compute the means were obtained from all evolution programme version pairs and represent potential problems in each category.

The red solid line shows the sum of the number of problems that had an impact on actual clients for the respective category. These values are normalised per number of clients affected. In other words, the sum of problems in each category was divided by the number of affected clients. Note that the variance values are relatively high and are therefore not plotted, see also Tables 2–4. As discussed in Section 4.1, the C2 category cumulates other atomic categories (M1, M2, MOD, ...) but several changes in atomic categories count as one change in C2. For that reason, the M2 and MOD values are higher than the C2 values in the graph.

It is interesting to see that while most changes that might cause problems are in M2 and MOD, while the category that does cause most problems on actual clients is M1 (and also C2, but this category includes M1). Moreover, problems in MOD had no impact on clients. Further research is needed to explain this pattern.

6.7. Compatibility and versioning

We also studied whether there is a correlation between versioning practises compatibility. In particular, "semantic versioning" [45] advocates that API-breaking changes should only occur when the major version number is changed.

For this purpose, we started with the six libraries that we found to have an impact on other programs (see Section 6.5). We excluded `jfreechart` and `informa` because we had only two and one versions in the dataset, respectively. The other four libraries were examined for correlations between version numbers and API-breaking changes.

Figs. 5–8 show the results for `lucene`, `hibernate`, `antlr` and `junit`. Bars in the graphs represent the number of incompatibilities summarised for all categories, calculated for each program version. They represent potential problems that happened in a respective version update. The solid red line summarises the number of incompatibilities which were propagated to clients, divided by the number of clients. An increase in this value means that new incompatibilities were propagated to clients in respective version updates, revealing that a potential problem has turned into an actual problem. A decrease means some incompatibilities were removed because clients have adapted to API changes. When the value remains constant, there is no change in the number of impacting incompatibilities. Within the given dataset, this was caused by the absence of a newer client versions in the corpus that has been modified to accommodate for the changes in the respective library updates.

This data allows us to answer **RQ4: Is there a pattern correlating API-breaking changes and versioning schemes?** We found no strong correlation between the versioning practises and the frequency of compatibility breaking changes. All programs we investigated had API-breaking changes when only the micro or build number increased. However, there is evidence that in most cases only major version changes correlate with actual impact on clients. Examples include `hibernate-4.0.0` or `lucene-3.0.0`. This might indicate that some communities have stricter versioning practises and move towards semantic versioning. On the other hand, `hibernate-3.6.0` is a case where compatibility affecting change is not aligned with adequate version number change.

The data also shows that the extent of change does not necessarily correspond with its effect on clients; this is similar to the results in Section 6.6. An example is (the minor version upgrade) `lucene-2.9.0` representing the second biggest change in this project, which does not increase the number of actual incompatibilities. A similar discrepancy occurs is in `junit-4.0`. This version is a radical redesign of junit to take advantage of annotations introduced on Java 1.5. However, the respective version upgrade shows only few incompatible changes, while later minor releases of the same project are considerably more unstable. Note that backward compatibility of `junit-4.0` was explicitly addressed by including `junit-3.*` legacy code (the `junit.framework` packages). It is likely that the unstable change that has occured in later minor version releases was caused by fine-tuning the new APIs in the `org.junit` packages.

These results indicate that while there is no strong pattern that correlates versioning practises with API stability, most projects

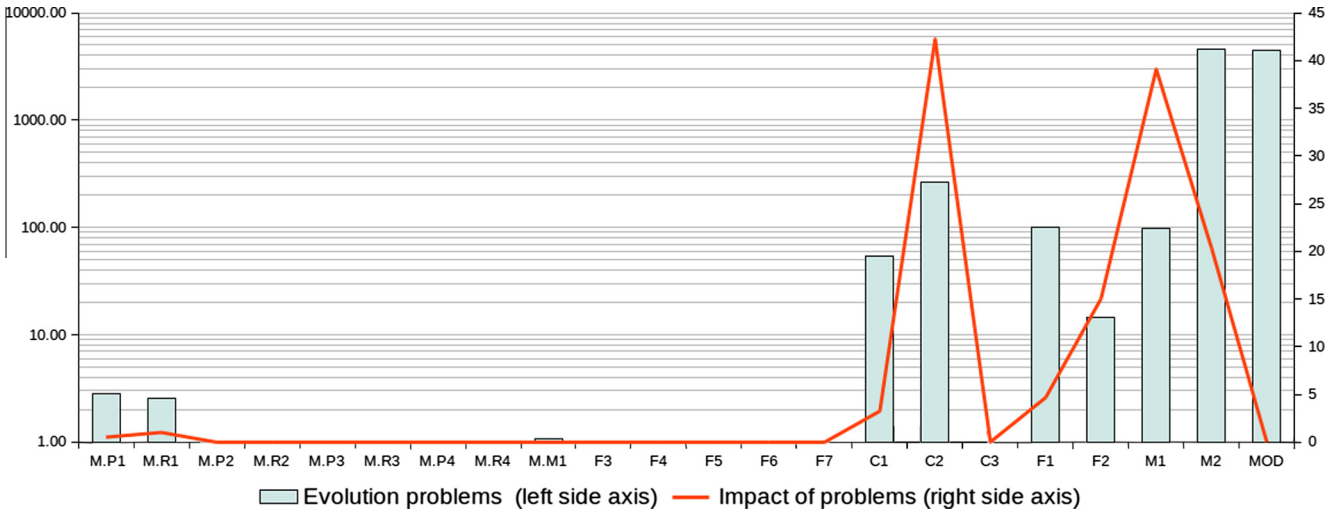

Fig. 4. Backward compatibility problems and their impact.

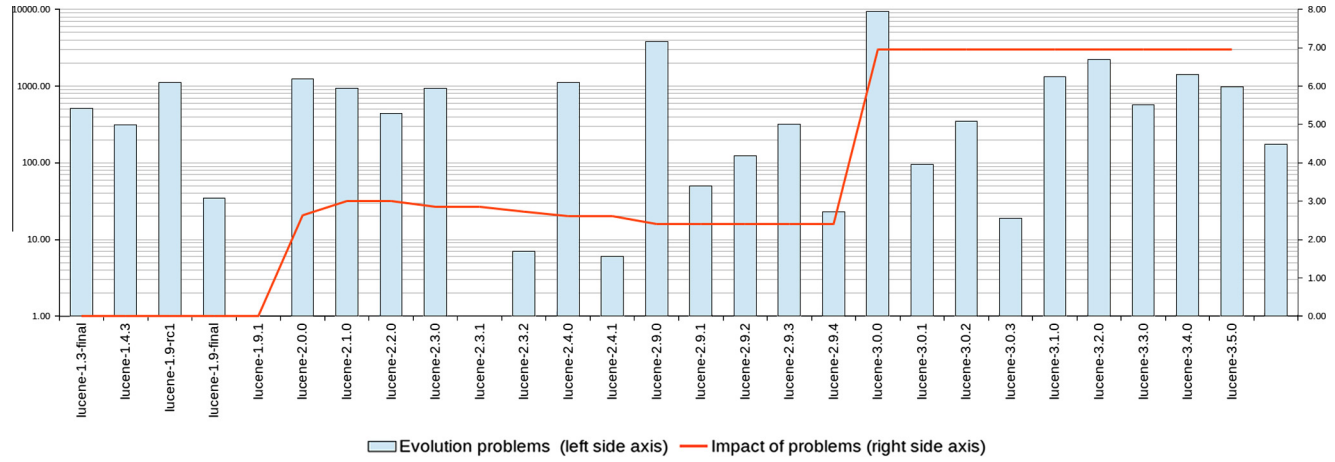

Fig. 5. Backward compatibility practice: lucene.

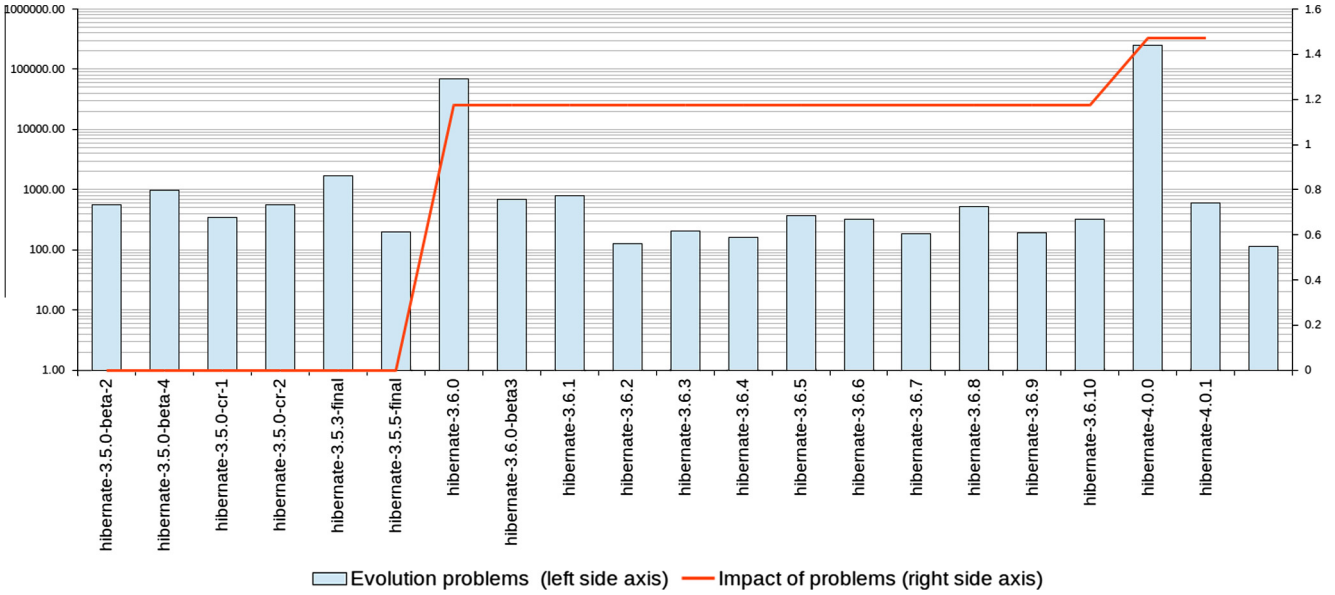

Fig. 6. Backward compatibility practice: hibernate (number of versions shortened).

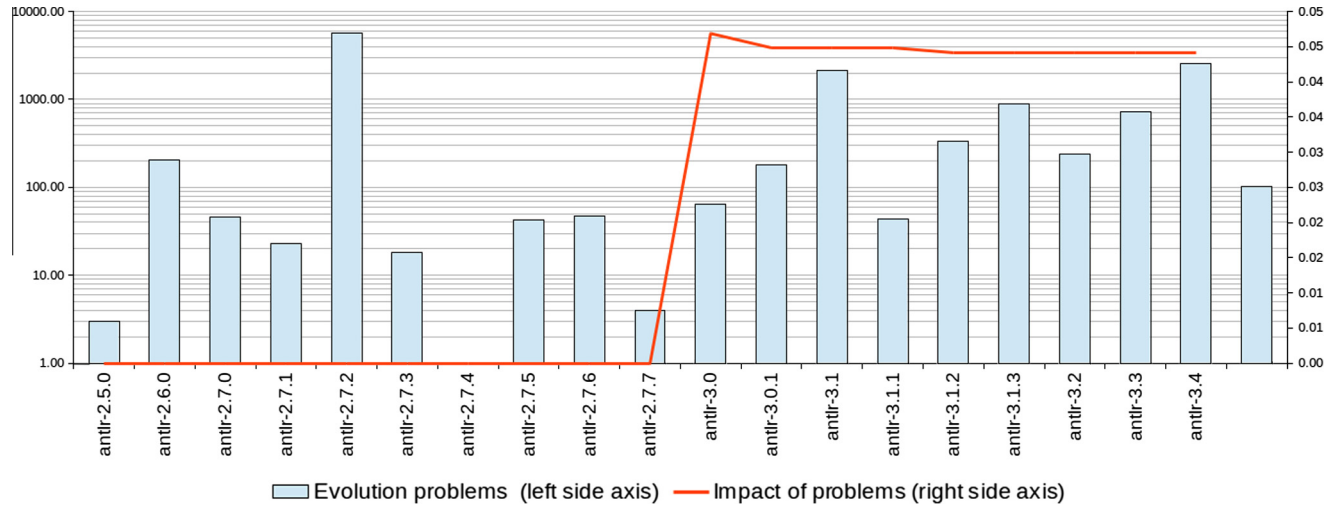

Fig. 7. Backward compatibility practice: antlr.

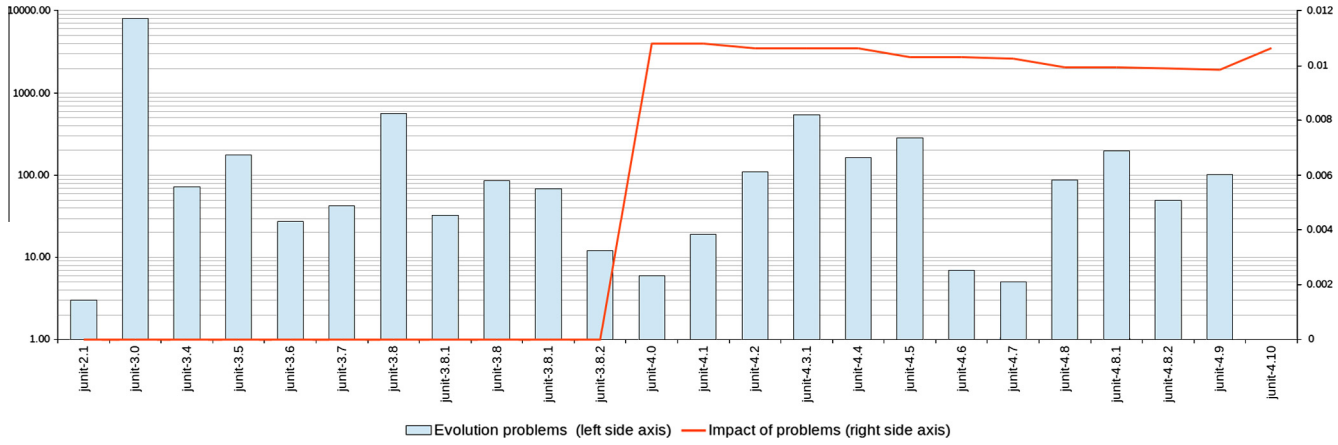

Fig. 8. Backward compatibility practice: junit.

seem to manage compatibility well in the sense that the impact on actual clients is confined to major version changes. This may indicate that many incompatible changes occur in parts of the API that are not considered to be public, though they are actually accessible. We must point out here that the number of clients in our data set was too small to provide statistical evidence and this observation needs to be supported by a more comprehensive study.

6.8. 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), *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`.

7. 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 such as whether a new version of a method could throw unchecked exceptions or simply change the way return 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 4.2 when the compiler fails to select the most specific method.

Finally, our method of cross-referencing programs using provided and used libraries described in 5.3 and 5.4 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 (so-called uber-jars). Another related issue is that sometimes the same class is provided by different libraries. We did encounter such a case as discussed in section 6.2. We did not systematically search for cases like this.

An assumption we make in our study is that clients do not use reflection on functionality provided by libraries and that neither providers not programs use runtime bytecode manipulation and creation. With reflection, clients can reason about the APIs of the classes they use. In particular, they can use reflection in conditional logic, and this makes each API change potentially behaviour-changing. With bytecode manipulation, unobservable changes could be introduced into the operations used by clients. The use of these techniques is hard or impossible to analyse using only static analysis. However, we assume that these assumptions will only have a marginal effect on the outcome of this study.

Some of the examined Maven projects contain so-called build profiles. These profiles allow developers to customise the build process, sometimes including or excluding some dependencies. Since there is usually no documentation for which profile to use, we always used the default profile. It does not necessarily mean that this is the profile used to build the final product. For this reason, we may have unintentionally used or missed some dependencies not used when the project is built.

8. Conclusions

In this paper, we have investigated the question what impact a library evolution has on API stability and how this affects programs. The short answer is that there are a lot of potential issues but only a few actual errors occur in the qualitas corpus dataset we studied.

After analysing 109 programs (each in several versions), we found that the *potential* for problems caused by unguarded evolution is high – as much as 80% of all version upgrades break vertical compatibility. However, we only detected very few *actual* problems where this affects horizontal (library linking) compatibility with actual programs: only 12 concrete client program versions are affected by incompatible changes in the libraries these programs include in their distribution and 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.

A surprising result is that using Maven with its automated dependency resolution increased dependency-related errors. While we found only one case where a project with embedded libraries had horizontal compatibility problems, we found 31 such cases in projects using Maven-defined dependencies. The impact analysis where we studied horizontal compatibility problems caused by evolution revealed a similar picture: there are significantly more problems in projects using Maven-defined dependencies (about 20–30 times higher in certain cases). In some cases, this was caused by multiple versions of libraries being used.

8.1. Answers to the research questions

The findings of this study can be summarised in the answers to the four research questions we set out to investigate.

RQ1: How frequent are API-breaking changes when programs evolve? API-breaking changes are very frequent (80% of version updates were detected as incompatible) and only a few versions are truly backwards compatible within the programs' complete evolution life-cycle.

RQ2: How do incompatible changes affect client programs – at build time during compilation or at link time? Most problems are caused by source incompatibilities, and clients are affected early at build time when compilation fails. Although problems causing only binary incompatibilities were relatively rare, examples at both class and field/method levels were detected. This shows that programs are also prone to run- and link-time failures.

RQ3: How many actual programs are affected by API-breaking changes? We did detect cases where incompatible changes broke actual client programs. This would have forced these programs to be recompiled and/or refactored. On the other hand, these changes tend to have little impact on projects with explicitly managed dependencies.

RQ4: Is there a pattern correlating API-breaking changes and versioning schemes? We found no strong correlation between the practice of assigning version numbers and the amount of compatibility issues in corresponding library versions. In particular, all programs we investigated had API-breaking changes when only the micro or build number increased. On the other hand, our experiments provided some evidence that suggests that changes in the major version number correlate rather well with actual impact on clients.

8.2. Implications

Despite not finding a large numbers of incompatibility problems, we did detect some potentially dangerous practices. We believe that applying some simple rules can significantly reduce the number of compatibility problems developers encounter.

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, there is evidence that most developers are only familiar with rules of source compatibility, and not aware of the subtle differences between binary and source compatibility [15]. 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, otherwise inlining can cause applications to rely on obsolete values. By "never" we do not only refer to "during the execution of the program", but rather "during the entire lifecycle of the program". Only the first part is enforced by the compiler and the JVM.

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 [36]. 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, Susan Eisenbach, Safe upgrading without restarting, in: Proceedings ICSM'03, 2003, pp. 129–137.

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

[3] Meriem Belguidoum, Fabien Dagnat, Formalization of component substitutability, Electron. Notes Theor. Comput. Sci. 215 (2008) 75–92.

[4] Antoine Beugnard, Damien Watkins, Jean-Marc Jézéquel, Noël Plouzeau, Making components contract aware, Computer 32 (7) (1999) 38–45.

[5] Joshua Bloch, Effective Java, Addison-Wesley Professional, 2008.

[6] Premek Brada, Enhanced type-based component compatibility using deployment context information, Electron. Notes Theor. Comput. Sci. 279 (2) (2011) 17–31. Proceedings FESCA'11.

[7] Premysl Brada, Specification-based component substitutability and revision identification, PhD thesis, 2003.

[8] Eric Bruneton, Romain Lenglet, Thierry Coupaye, ASM: a code manipulation tool to implement adaptable systems, Adaptable Extensible Component Syst. 30 (2002).

[9] John Corwin, David F. Bacon, David Grove, Chet Murthy, MJ: a rational module system for Java and its applications, SIGPLAN Not. 38 (11) (2003) 241–254.

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

[11] Brad J. Cox, Object Oriented Programming: An Evolutionary Approach, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[12] Ilie Şavga, Michael Rudolf, Refactoring-based support for binary compatibility in evolving frameworks, in: Proceedings GPCE '07, ACM, New York, NY, USA, 2007, pp. 175–184.

[13] Jim des Rivières, Evolving Java-based APIs, 2007. <http://wiki.eclipse.org/Evolving_Java-based_APIs> (accessed 28.08.13).

[14] Jens Dietrich, Kamil Jezek, Premek Brada, Broken promises: an empirical study into evolution problems in Java programs caused by library upgrades, in: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), IEEE Computer Society, 2014, pp. 64–73.

[15] Jens Dietrich, Kamil Jezek, Premek Brada, What java developers know about compatibility, and why this matters, 2014. <http://arxiv.org/pdf/1408.2607v1.pdf>.

[16] Jens Dietrich, Lucia Stewart, Component contracts in eclipse – a case study, in: Proceedings CBSE'10, Springer, 2010, pp. 150–165.

[17] Danny Dig, Ralph Johnson, How do APIs evolve? A story of refactoring, J. Softw. Maintenance Evol.: Res. Pract. 18 (2) (2006) 83–107.

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

[19] Mikhail Dmitriev, Language-specific make technology for the Java programming language, in: Proceedings OOPSLA '02, ACM, New York, NY, USA, 2002, pp. 373–385.

[20] Sophia Drossopoulou, David Wragg, Susan Eisenbach, What is Java binary compatibility?, ACM SIGPLAN Notices, vol 33, ACM, 1998, pp. 341–361.

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

[22] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex. Buckley, The Java™ Language Specification, seventh ed., Oracle, Inc., California, USA, 2012.

[23] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, Maximilian Irlbeck, On the extent and nature of software reuse in open source Java projects, in: Klaus Schmid (Ed.), Top Productivity through Software Reuse, LNCS, vol. 6727, Springer, Berlin, Heidelberg, 2011, pp. 207–222.

[24] JBoss, Inc. JBoss Project Versioning, 2012. <https://community.jboss.org/wiki/JBossProjectVersioning> (accessed 02.09.13).

[25] Kamil Jezek, Jens Dietrich, On the use of static analysis to safeguard recursive dependency resolution, in: Proceedings EUROMICRO DSD/SEAA 2014, 2014.

[26] Kamil Jezek, Lukas Holy, Premek Brada, Supplying compiler's static compatibility checks by the analysis of third-party libraries, in: Proceedings CSMR '13, IEEE Computer Society, 2013, pp. 375–378.

[27] Ralph Keller, Urs Hölzle, Binary component adaptation, in: Proceedings ECOOP '98, Lecture Notes in Computer Science, vol. 1445, Springer, 1998, pp. 307–329.

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

[29] M.M. Lehman, L.A. Belady (Eds.), Programme evolution: processes of software change, Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[30] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex. Buckley, The Java™ Virtual Machine Specification – Java™ SE, seventh ed., Oracle, Inc., 2012.

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

[32] Robert C. Martin, The dependency inversion principle, C++ Rep. 8 (6) (1996) 61–66.

[33] Robert C. Martin, The interface segregation principle: one of the many principles of OOD, C++ Rep. 8 (1996) 30–36.

[34] Tom Mens, Juan Fernández-Ramil, Sylvain Degrandsart, The evolution of eclipse, in: Proceedings ICSM'08, October 2008, pp. 386–395.

[35] Oracle, Inc., Java ™ Product Versioning, 2013. <http://docs.oracle.com/javase/7/docs/technotes/guides/versioning/spec/versioning2.html> (accessed 28.08.13).

[36] Oracle, Inc., Project Jigsaw, 2013. <http://openjdk.java.net/projects/jigsaw/> (accessed 28.08.13).

[37] OSGi Alliance, Semantic Versioning Technical Whitepaper Revision 1.0, 2010. <http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf> (accessed 28.08.13).

[38] OSGi Alliance, OSGi Service Platform Release 4.3, 2012. <http://www.osgi.org/Release4/Download> (accessed 28.08.13).

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

[40] Steven Raemaekers, Arie van Deursen, Joost Visser, An analysis of dependence on third-party libraries in open source and proprietary systems, in: Sixth International Workshop on Software Quality and Maintainability, SQM, vol. 12, 2012.

[41] Steven Raemaekers, Arie van Deursen, Joost Visser, Measuring software library stability through historical version analysis, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 378–387.

[42] Steven Raemaekers, Arie Van Deursen, Joost Visser, Semantic versioning versus breaking changes: a study of the maven repository, Technical report, Delft University of Technology, Software Engineering Research Group, 2014.

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

[44] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, James Noble, The qualitas corpus: a curated collection of Java code for empirical studies, in: Proceedings APSE C'2010, IEEE, 2010, pp. 336–345.

[45] Tom Preston-Werner, Semantic Versioning 2.0.0, 2014. <http://semver.org/> (accessed 20.06.14).

[46] Terry Winograd, Beyond programming languages, Commun. ACM 22 (7) (1979) 391–401.