Information and Software Technology xxx (2015) xxx–xxx

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 xxxx

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

Please cite this article in press as: K. Jezek et al., How Java APIs break – An empirical study, Inform. Softw. Technol. (2015), http://dx.doi.org/10.1016/j.infsof.2015.02.014