API Evolution Data Corpus and Tools Challenge

Kamil Jezek
Department of Computer Science and Engineering
NTIS – New Technologies for the Information Society
Faculty of Applied Sciences, University of West Bohemia
Pilsen, Czech Republic
kjezek@kiv.zcu.cz

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

***Abstract*—Development of independently released software components is nowadays widely supported by tools. Among other benefits, the tools help guarantee backward compatibility of new versions of components. In other words, absence of non-breaking changes depends on ability of the tools to detect them. But do the tools really cope well? This work aims at answering this question. In brief, we have selected tools that analyse syntactic API changes and benchmarked them. We provide results together with the synthetic data used for the test. The data set should be itself a valuable contribution as it may test future tools.**

I. Introduction

Quality assurance has been long understood as integral part of software development. While exercising programs either manually or by automated tests is still common, more effective approaches come to light. The main rationale is to statically inspect the code and detect possible problems without costly invoking a program. Back in the beginning of millennia Jacobs however pointed at challenges of static verification [10] and claimed:

> ”While abstract formalisms were useful at the time for explaining the main ideas in this field, they are not very helpful today when it comes to actual program verification for modern programming.”

Despite obstacles, both source and byte-code analysis started to be popular for Java program verifications. Over the last two decades a lot of approaches to byte-code verification have followed up. Let us name Leroy [16] who reviews byte-code verification techniques which mainly concentrates on security and byte-code consistency issues. He was followed by others Male [18], Klein [15] or Burdy [3] providing various byte-code based verifications. Noticeably, quite a few works [11], [12], [21], [22], [7] target API evolution in standard Java, but also in Android [17].

Approximately at the same time, static verification implemented in open source tools started to appear. Notoriously known are source-code style checkers such as `PMD`[1], `Checkstyle`[2], or `Findbugs`[3] that are widely used and integrated into development tools or continuous integrations such us `Jenkins`[4] or `SonarQube`[5].

[1]https://github.com/pmd
[2]http://checkstyle.sourceforge.net/
[3]http://findbugs.sourceforge.net/
[4]https://jenkins.io/
[5]http://www.sonarqube.org/

Another set of tools are checkers of API evolution. They are generally useful for assessing compatibility of independently evolved libraries. They help developers assure smooth process of upgrading systems or releasing new versions of components. In contrast to the previous group, they work usually with byte-code and thus ideally fit for inspection of third-party binary libraries. All tools we are aware of are included in this work and listed in Table I. They were collected putting together our knowledge, cross-refrences and search through developer forums, mostly stackoverflow.com.

missing JDiff

To our best knowledge, these tools are less popular and also seem to be surrounded by smaller communities in comparison to the tools that check source-code style[6]. It is interesting observation as API compatibility is not less important then code-style violations, or at least should not be. This work does not aim at answering if and why the tools are less popular, but it should provide insight into their quality and ability to detect API defects.

extend why we need a new dataset

In this work, we additionally propose a dataset simulating syntactic changes of API, which has been used to benchmark nine tools.

The contribution of this work is twofold. First, it provides an extensible dataset with possibly wide applications such as mock-testing, replication studies, or benchmark of new tools, etc. Secondly, we benchamrk how successfully nowadays tools deal detect incompatibilities.

finish

Remainder of this paper is structured as follows ...

II. Related Work

In the technical domain, the term compatibility denotes[7] the “ability to be used together” and “designed to work with another device or system without modification”. Various definitions of compatibility related to software components exist, both in research [4], [1], [24], [2] and technical [8], [20],

[6]One of the tool, `Clirr`, has three members, one contributor and finally is not developed since 2005. Still active is `Revapi` but with only four contributors. Bigger seems to be `japicmp` with 11 contributors. But they cannot match with `checkstyle` wiht 92 contributors or `PMD` with 58
[7]Source: the Merriam-Webster dictionary.

[19] literature, mostly dealing with the issue of either correct replacement or interoperability. ... TODO sophisticated compatibility handling aka: non-functional properties, behaviour, models, etc.

...

Overview of possible API breaking changes is collected in [5]. ...

Belguidoum [1] distinguishes between *vertical* and *horizontal* compatibility which may be respectively paraphrased as backward and client-provider compatibility. Vertical compatibility plays role when vendors desire to produce backward compatible libraries, which allows for smooth system updates. On the other hand, horizontal compatibility plays role in checking system composition. Both directions should be taken into account to successfully adapt "TODO.precise.cit: composition of a system from trhid-partly independently evolved components" defined in last decade by Szyperski [23].

...

III. Background: About Compatibility

Consequence of a wrongly selected library is potential incompatibility with its client. The notion of compatibility is complex as every modification of a library may influence the way other libraries can use, interact, extend, observe or substitute it, in various ways. However, practical applications usually rely on syntactic changes expressed via API. In other words, a syntactic change in API that does snot prevent client from linking or compiling is expected as compatible, even if it can have impact on behaviour. For instance, while a change from `List` to `Set` is acceptable for assignment to a field typed to `Collection`, the same change may impact clients that rely on particular elements order.

The Java Language Specification formally defines acceptable changes in respect to binary compatibility [9, ch. 13] as:

> "a set of changes that developers are permitted to make to a package or to a class or interface type while preserving (not breaking) compatibility with pre-existing binaries."

The rules are strictly defined with respect to the static analysis performed during linking, which significantly differs from the notion of source compatibility, which is checked by the compiler as the consistency between a program and a library. For this reason, the specification explicitly recommends:

> "tools for the Java programming language should support automatic recompilation."

In the same chapter it, however, admits that

> "it is often impractical or impossible to automatically recompile the pre-existing binaries that directly or indirectly depend on a type that is to be changed."

When a program is built and deployed, a mixed notion of compatibility is used. As the program is compiled, the source compatibility with the libraries is checked by the compiler. The binary compatibility is checked instead when the program is invoked. When not all directions of compatibility are satisfied, situations where a system may be compiled but cannot run or vice-versa may appear.

Compatibility also depends on how a library is used, i.e. if a library is used (invoked) only or also implemented (used for extension in sense of the object-oriented paradigm). For instance, a method added to an interface is acceptable for the client invoking this interface, but breaks compilation for someone implementing that interface as all methods must be implemented[8].

There were already works to catalogue API changes [5] and a wide set of empirical studies [14], [6] or two works by Raemakers [21], [22] to measure extend of the problem. Recently, we proposed even an approach to mitigate big part of the problem by employing run-time adaptation into Java [13]. However, current tools have not bee obsoleted yet, as this research prototype is still waiting for its adoption by industry.

IV. API Changes Data Corpus

Test data proposed in this work composes a corpus of possible syntactic API changes, which model evolution and releases of consequent versions of a library. The data are separated into eight top level categories: access modifiers, data types, exceptions, generics, inheritance, class members, other (non-access) modifiers and borderline (uncategorised) category. Each category is filled with examples of API changes, which were basically constructed following subsection is section 13 in the Java Language Specification, a catalogue created by Rivieres [5] and a few more sources[9]. The data does not have to be complete as some cases did not have to be know in the time, or new cases may appear as the language evolve. For this reason, the corpus is extendable and new examples may be added.

The corpus is split into three directories: `lib-v1`, `lib-v2` and `client`. As the names suggest, the directories contain a first (original) version a library, a second (evolved) version of a library and a client application, which invokes the library. The directories model real-life scenarios where a client uses libraries, but the libraries in the corpus are simplistic and contain only API with dummy implementations. The client application shows possible invocation of a library, thought it does not have to be exhaustive and more cases will actually exist.

Each library as well as the client hold a set of subdirectories, Java packages, representing concrete API evaluation example. The package names are constructed following way:

```
<category><element><change>
```

Where `category` is one of the eight categories, `element` is a representation of a changed element (class, method, field, ...) and `change` describes content of the change. For instance, a case named `dataTypeClassFieldBoxing` means that a class field changed its data types and the concrete change was boxing.

[8]Exceptions are Java 8 `default` methods

[9]Several developer forums such as stackoverflow.com

The design based on the naming convention allows for extensibility of the corpus by simply adding new cases to sub-directories (packages) following the convention. In fact, the convention is not enforced, but recommended to keep order in the relatively big number of data – the corpus currently contains 251 examples.

The corpus is provided in the form of source-code with an `ant` script to build binaries. It may be invoked simply by typing `ant` from the command-line. The script output are three JAR files named the same way as the original source directories.

The whole structure of the corpus looks as follows (<> is shortcut for the `<category><element><change>` triplet described above):

```
<root>
  +- client/src/<>/Main.java
  +- lib-v1/src/lib/<>/<>.java
  +- lib-v2/src/lib/<>/<>.java
  build.xml
  compatibility.sh
```

Although the corpus described so far may be used as such, we provide additional meta-data. They must be generated before first usage to remain up-to date when the corpus is extended. A linux bash script named `compatibility.sh` stored in the corpus root is provided to do this. It generates a simple CSV file with three columns respectively listing: the name of a change as described above, and two columns informing about source and binary compatibility. Value "1" is printed for a compatible change, "0" otherwise.

The script generates metadata by following steps, it:

1) reads all changes stored in the `client` directory
2) compiles `lib-v1` and `lib-v2` directories,
3) compiles the client against `lib-v1.jar` – it should succeed,
4) compiles the client against `lib-v2.jar` to check source compatibility– it may fail,
5) invokes the client originally compiled against `lib-v1.jar` with `lib-v2.jar` to check binary compatibility – it may fail,
6) writes result ("1/0") to the CSV file.

Steps 4 and 5 fail when a respective change is source or binary incompatible, which is in the last step written into the CSV file.

We make the corpus publicly available as a GitHub project for replication studies, benchmark of new tools or similar:

```
https://github.com/kjezek/
api-evolution-data-corpus/
```

Following sub-sections detail the corpus separated into categories. Short discussions is provided in each sub-section to overview why the category is signification.

A. Data Types

Data types represent fundamental changes used for method and constructor signatures, field types, generics, inheritance and exceptions.

We cover changes respecting general features of object-oriented paradigm, i.e. polymorphisms as well as changes particular to Java. The basic changes are:

- Del – a type is removed.
- Inst – a type is added.
- Gen – generalised, i.e. type is moved up in an inheritance hierarchy. For instance, a `java.lang.Number` is generalisation of `java.lang.Integer`.
- Spe – specialised, which is opposite of the previous case.
- Mut – mutated, a type is changed to an incompatible one, i.e. not coming from the same inheritance tree.

Explicit cases to cover primitive types are modelled as well:

- Narrow – a "specialising" conversion for primitive types. E.g. `long` changed to `int`.
- Widen – the opposite to the previous case.

Finally, Java allows for two more conversion to simplify work with primitive and wrapper types:

- Box – a primitive type is converted to its wrapper type, e.g. `int` to `java.lang.Integer`.
- Unbox – a wrapper type converted to a primitive type.

Changes in this category play a different role for source and binary compatibility. In particular, `Gen`, `Spe`, `Narrow`, `Widen`, `Box` and `Unbox` are conversions performed only by the Java compiler, not the linker. It means, that they are always binary incompatible, but may be source compatible depending on usage. For instance, `Gen` is a source compatible conversion for a method parameter type, while `Spe` is compatible for a method return type. Example is a method that used to accept `java.lang.Integer` as a parameter type. It will remain compatible when it is changed to `java.lang.Number`. The opposite case holds for return types where the return type may be only specialised. However, the compiler decides correct conversions that s compiled into the byte-code and no more conversions are performed by the linker.

Changes `Del` and `Mut` are neither source nor binary compatible as no fall-back mechanism for non-existing or incompatible types is provided in Java.

B. Exceptions

Java distinguishes between so called checked and unchecked exceptions. The checked ones must be handled by a client code, either by propagation to upper levels or managed by the `try-catch` block. The unchecked exceptions are propagated automatically and may be optionally caught as well.

It is less known that the handling of exceptions in the client code is checked only by the compiler, not the linker. As a consequence, changes in exceptions impact only source compatibility. When a library method is updated so that it adds a new checked exception, the original client code cannot be compiled and it must be refactored to accommodate proper exception handling. On the other hand, if the same library is used in conjunction with an already compiled client, they will link smoothly together.

The corpus combines examples of specialised, generalised or mutated exception types in variants for checked and unchecked ones.

C. Generics

Generics were added to Java relatively late in version 1.5 with strong regard to compatibility with previous Java version. Their employment had required several simplifications, most noticeably so called *erasures*. Generics are erased during the compilation from the caller site and their persist only on the target side. When the linker searches for types, it works as if only raw types without generics were used.

Whereas a client code is checked by the compiler to correctly use generic types, a binary code that uses generics may be combined with the code not using generics thanks to erasures. The impact to compatibility is evident. A lot of changes that are binary compatible do not have to be source compatible.

A typical example are bounds of a parametrized type: for instance, if a list is defined as `java.util.List<String>` only `String` values may be added and the compiler checks it. However, when the definition is changed to `java.util.List<Number>` and only binaries updated, the system will successfully link. Let us note, that such a program will likely later rise `ClassCastException` or similar as erasures are effectively replaced by the `checkast` byte-code instruction.

D. Inheritance

Add citations - Tulach?

Inheritance is sometimes not considered at all in respect to compatibility and some best practices discourage implementations/extensions of API classes [**?**]. The reason is that API must be almost frozen to be backward compatible also for inheritance.

Some changes such as removed methods clearly break compatibility, but some breaking changes are less evident. Addition of an method to an interface is still a quite known example, but a possible compatibility issue of making a method more visible is, however, less obvious. In detail, changing visibility e.g. from `private` to `public` may seem harmless as only more is provided, but the new public method may overlap with the same method in a sub-class. When the sub-class method has a stricter access, the compilation also fails as the access cannot be weakened through inheritance. As a result, adding a public method to a class may break compatibility for inheritance.

Since the changes possibly impacting inheritance are partly covered by other categories (method, modifier, types etc. changes), this category contributes to the corpus with several more examples with class/interface definitions modified in a sub-class. Several examples with methods moved up and down in the hierarchy tree are provided as well.

E. Members

Members are elements defined in a Java class including fields, methods and constructors. This category contains examples of removed or added members that have expected impact on compatibility: removed elements are always incompatible, added elements may be incompatible for inheritance as discussed above.

The problem of added abstract/interface methods addressed in Java 1.8 by `default` methods is modelled in the data set as well.

F. Access Modifiers

Access modifiers may be either weakened or strengthened and may be applied to a constructor, a method, a field, a class and an interface. These combinations are reflected in the corpus.

A change making an element more accessible is usually a compatible change while restricting the access is incompatible. It does not differ for source and binary compatibility. The only known exception is inheritance of a method with a weaker access as already discussed.

G. Other Modifiers

Other, non-access, modifiers have various purposes in Java and for this reason have different impact to compatibility. Modifier `volatile`, `transient`, `native` or `strictfp` are denoted to specific treatment of respective elements, `final` or `abstract` are used in conjunction with inheritance, and `static` deals with access context. Sometimes one modifier is used for multiple purposes, e.g. constants are implemented as `final` fields, while `final` also denotes classes that cannot be inherited.

There is no pattern how these modifiers impact compatibility. For instance, a tagging modifier `transient` does not break compatibility while `native` does. It is because `native` requires a special treatment by JVM while `transient` is only a meta-information. Modifiers `final` and `abstract` have obvious effect as their adding or removing may break inheriting classes.

Interesting is the `static` modifier that may be in certain cases source compatible and binary incompatible. It is more discussed in [14, Section 4]: static elements, fields and methods, may be invoked from non-static context (via a reference) and pass the compilation. However, this combination is forbidden at runtime as different byte-code instructions are generated for static and non-static access and thus it fails when the byte-code is updated without recompilation.

H. Borderline Cases

Java contains several specific features that are grouped here. Notoriously know is implicit inheritance of `Object` by any classes. Maybe less known is that any Java array by default implements `Cloneable` and `Serialisable`. Consequence is that possible specialisation between user classes and these classes must be taken into account when deciding compatibility.

Another example in this category is a change from class to interface or vice-versa also detailed in [14, Section 4]. It is interesting because invocation of methods does not differ between class and interface methods in the source-code. However, different byte-code instructions are generated leading to another discrepancy between source and binary compatibility. In other words, when a library class is changed to an interface or vice-versa, the client invoking methods from this interface/class may be recompiled, but cannot be linked without recompilation.

V. Tools Challenge

We searched for tools that are capable of discovering API syntactic backward compatibility and included them into benchmarkk. They are listed in Table I together with basic information about their authors, current versions, licensing and basic usages. All these tools were benchmarked to find out how they cope with finding incompatibilities, results are provided below.

A. Methodology

The tools were tested using following approach: we generated the meta-data CSV file as described in Section IV, then we removed lines representing only compatibilities. After that, we invoked all the tools and redirected their output to text files. In certain cases we removed lines from outputs representing a compatible change. Finally, we iterated the meta-data CSV file and used string matching to find changes in tools outputs. We collected the results in another CSV file, which lists changes in lines and tools in columns. The columns contain "1" for a correctly detected incompatibility and "0" meaning that a tool did not find the incompatibility.

The lines with compatible changes were removed from the original CSV meta-data to prevent false negatives. We had to analyse only provably incompatible changes because the current client does not have to handle all possible invocations. An incompatible usage not modelled in the client may exist and if detected by a tool, it would not match with the expected compatibility leading to a wrong result.

Compatible changes were removed from the tools outputs to prevent false positives. Some of the tools list all API they crawled with additional compatibility classification. It could cause an incorrect string matching if a change not recognised as incompatible were listed in the output. This step differs for each tool. Some of the tools output only incompatibilities and do not have to be corrected (`japichecker`, `japicc`) while some have to. Usually such lines can be easily caught by a simple regular expression as they contain representative strings such as char `!` (`japicmp`), text `100% Compatible` (`japitool`), `NON_BREAKING` (`revapi`), `INFO` (`clirr`) etc.

B. Extendability

The whole process is automated and may be invoked by a bash script `./benchmark.sh`. The script prepares the meta-data, invokes the tools, corrects outputs and analyse results. It delegates invocation of the tools to the script `tools/run.sh`, which executes all tools one-by-one.

For instance, the `run.sh` script contains following lines to invoke `japicmp`:

```
REPORTS=".reports"
java -jar japicmp/japicmp-0.7.2.jar \
  -o ../lib-v1.jar \
  -n ../lib-v2.jar \
  -a private > "$REPORTS"/japicmp.txt

grep -v '===  UNCHANGED' \
  "$REPORTS"/japicmp.txt > japicmp.txt.tmp
mv japicmp.txt.tmp "$REPORTS"/japicmp.txt
```

New tools may be added to the benchmark simply by adding invocable lines to this script so that they produce a textual output stored in the `tools/.reports`.

The structure of the corpus including the tools benchmark looks like:

```
<root>
  +- client/src/<>/Main.java
  +- lib-v1/src/lib/<>/<>.java
  +- lib-v2/src/lib/<>/<>.java
  +- tools/.reports
  +- tools/<tool>
  +- tools/run.sh
  build.xml
  compatibility.sh
  benchmark.sh
```

C. Results

The result of described experiment has shown that the tools widely differ in their ability to find compatibility breaking changes. Results are provided in Table II as percentages of successfully detected compatibility breaking changes. They are separated to categories with a summary in the last row.

While the results show clearly that the worst tool is `clirr` and the best is `sigtest`, detailed analysis reveals that `clirr` may be a better choice than some of the better performing tools in certain use-cases.

`Clirr` is not actively developed since 2005, and evidently does not recognize generics and exceptions. However, it works well in other categories and may be still useful for detecting only binary incompatible changes as generics and exceptions impacts only srouce compatibility. Similar situation appears for `japicmp` with overall a poor result, which is however caused by unsporting generics and a few bugs in detecting modifiers. Otherwise, the tool works well.

The second place is occupied by `japitool` which also seems to be no more developed since 2006, thought still available as part of Linux distributions (Debian/Linux in particular).

Tools such as `japicc` or `revapi` showed overall better score, but they have several issues scattered among categories. They may be less reliable in production as they can miss important issues in both categories of source and binary compatibility, while worse performing tools like `clirr` or `japicmp` may be a better choice when source compatibility is out of interest. Nonetheless, both tools are still developed and may be thus improved in the future.

Tool	Clirr	Japicmp	japiChecker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
Basic info									
Author	Lars Khne	Martin Mois	William Bernardet	Andrey Pono-marenko	Lukas Krejci	Oracle	Stuart Ballard	Vlad Skarzhevskyy	UWB
License	LGPL	A2.0	A2.0	LGPL	A2.0	GPLv2	GPL	LGPL	ask
Version	0.6.0	0.7.2	0.2.1	1.5	0.4.2	3.1	0.9.7	2.0.3	1.0.9
Release	9/27/2005	3/20/2016	10/3/2015	4/8/2016	3/30/2016	4/8/2016	11/13/2007	12/12/2008	
Output									
TXT	yes	yes	yes		yes	yes	yes	yes	yes
XML	yes	yes							
HTML	yes	yes		yes					
Integration									
CLI	yes	yes	yes	yes	yes	yes	yes	yes	
Maven	yes	yes	yes		yes	yes		yes	yes
Ant	yes		yes		yes	yes			
libray		yes							yes

TABLE I
TESTED TOOLS (GPL//LGPL = GNU GPL/LGPL, A2.0 = APACHE 2.0)

`Sigtest` wins the benchmark as it is able to detect almost all problems. It fails only in two changes, detection of the removed `strictfp` modifier and addition of the `native` modifier, which are both binary incompatible. As they are very specific modifiers, we do not expect their frequent changes among library versions. For this reason, `sigtest` may serve as the most reliable tool from this benchmark.

Table III provides insight into results separated for source and binary incompatibilities. First line shows changes that are only source incompatible and binary compatible. The second line in contrast lists changes that are binary incompatible, but may be either source compatible or incompatible. We separated the data this way to test the tools specifically for source compatibility and for the rest.

The table provides interesting results, the tools perform much better in detecting binary incompatibilities. Worst in this category is `revapi` while most of the tools detected more than 90% of issues. On the other hand, the tools lack ability to detect source incompatibility. The only actually reliable tools are `sigtest` and `japitool` that correctly recognised all source incompatibilities. Partly useful is `revapi` with about 88% of successful results. Other tools detected only a small number of problems and cannot be recommended.

Non-functional properties such as easiness of usage is nonetheless important in recommending the tools. A tool with a few bugs may be a better choice if it provides a better user comfort. While we did not measured it systematically in this work, we did some observation about tools usage and output. It must be said that the tools are very similar in this aspect. All provide CLI with a few options to input JAR files and print out a human readable formatted TXT output. We see non of the formats particularly better then another one. One exception is `japicc` which provides a HTML output classifying severity of changes, which may be a better readable by humans and especially by non-programmers.

To summarize, the experiment shows that the most usable tool is `sigtest`, which is distributed as open-source and may be easily integrated into development process via CLI, Maven or Ant plugin. Furthermore, it was detected that other tools are in reality usable only for checking binary compatibility. Nonetheless, it may be sufficient in many scenarios as library updates are usually distributed in binary forms. Hence, binary compatibility checking may help find the most unpredictable runtime failures caused by opaque third-party libraries. Although a source incompatible change may break a system as well, it is detectable by project build early in the development phase and thus it is less harmful.

D. Threats to Validity

- data completeness - bugs in data analysis

VI. CONCLUSION

The conclusion goes here.

ACKNOWLEDGMENT

The authors would like to thank...

REFERENCES

[1] Meriem Belguidoum and Fabien Dagnat. Formalization of component substitutability. *Electronic Notes on Theoretical Computer Science*, 215:75–92, June 2008.

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

[3] Lilian Burdy and Mariela Pavlova. Java bytecode specification and verification. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1835–1839, New York, NY, USA, 2006. ACM.

[4] Carlos Canal, Ernesto Pimentel, and José M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2):105–138, October 2001.

[5] Jim des Rivières. Evolving Java-based APIs. http://wiki.eclipse.org/Evolving_Java-based_APIs. [Accessed: Dec. 1, 2014], 2007.

[6] Jens Dietrich, Kamil Jezek, and Premek Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *IEEE CSMR-WCRE Software Evolution Week*. IEEE Computer Society, 2014.

[7] S. A. Ebad and M. A. Ahmed. Measuring stability of object-oriented software architectures. *IET Software*, 9(3):76–82, 2015.

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

[9] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. California, USA, java se 7 edition edition, February 2012.

Category	clirr	jacc	japicc	japiChecker	japicmp	japitool	jour	revapi	sigtest
Access Modifiers	100.00%	100.00%	83.33%	100.00%	100.00%	100.00%	83.33%	83.33%	100.00%
Data Types	100.00%	100.00%	89.36%	100.00%	100.00%	100.00%	100.00%	95.74%	100.00%
Exceptions	0.00%	0.00%	100.00%	100.00%	100.00%	100.00%	100.00%	71.43%	100.00%
Generics	0.00%	33.33%	5.88%	0.00%	0.00%	100.00%	17.65%	100.00%	100.00%
Inheritance	71.43%	100.00%	71.43%	85.71%	100.00%	100.00%	100.00%	42.86%	100.00%
Members	100.00%	100.00%	84.21%	89.47%	100.00%	100.00%	84.21%	42.11%	100.00%
Other Modifiers	61.54%	84.62%	84.62%	53.85%	84.62%	69.23%	76.92%	61.54%	84.62%
Others	100.00%	100.00%	75.00%	100.00%	100.00%	100.00%	100.00%	50.00%	100.00%
Total	57.79%	72.08%	59.74%	61.04%	65.58%	97.40%	68.18%	82.47%	98.70%

TABLE II
CORRECTLY DETECTED INCOMPATIBILITIES

Type	clirr	jacc	japicc	japiChecker	japicmp	japitool	jour	revapi	sigtest
Source	13.24%	41.18%	25.00%	20.59%	25.00%	100.00%	38.24%	88.24%	100.00%
Binary	93.02%	96.51%	87.21%	93.02%	97.67%	95.35%	91.86%	77.91%	97.67%

TABLE III
SOURCE VS BINARY INCOMPATIBILITIES

[10] Bart Jacobs, Joseph Kiniry, and Martijn Warnier. *Java Program Verification Challenges*, pages 202–219. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[11] K. Jezek and J. Ambroz. Detecting incompatibilities concealed in duplicated software libraries. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pages 233–240, Aug 2015.

[12] Kamil Jezek and Jens Dietrich. On the use of static analysis to safeguard recursive dependency resolution. In *SEAA 2014 [in print]*, 2014.

[13] Kamil Jezek and Jens Dietrich. Magic with Dynamo – Flexible Cross-Component Linking for Java with Invokedynamic. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 12:1–12:25, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Kamil Jezek, Jens Dietrich, and Premek Brada. How java apis break - an empirical study. *Journal of IST*. submitted to second review.

[15] G. Klein and M. Wildmoser. Verified bytecode subroutines. *Journal of Automated Reasoning*, 30(3):363–398, 2003.

[16] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *Journal of Automated Reasoning*, 30(3):235–269, 2003.

[17] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[18] Chris Male, David J. Pearce, Alex Potanin, and Constantine Dymnikov. *Java Bytecode Verification for @NonNull Types*, pages 229–244. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[19] Oracle. Kinds of compatibility. Online: https://blogs.oracle.com/darcy/entry/kinds_of_compatibility (Jan, 2015).

[20] The OSGi Alliance. *Semantic Versioning: Technical Whitepaper*, revision 1.0 edition, May 2010.

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

[22] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Proceedings of the 2012 IEEE International Conference on Software Maintenance (ICSM)*, ICSM '12, pages 378–387, Washington, DC, USA, 2012. IEEE Computer Society.

[23] Clemens Szyperski. *Component Software, Second Edition*. ACM Press, Addison-Wesley, 2002.

[24] Richard N. Taylor, Nenad Medvidovic, and Eric Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.