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, non-breaking changes in components depend on ability of the tools to detect these changes. But how the tools cope? In this work, we have selected tools that analyse syntactic compatibility of API changes and tested how they perform. We provide results together with the synthetic data used for the test. The data set is itself a valuable contribution of this work as it may be used for testing future tools.**

I. Introduction

...

In this work, we propose a dataset simulating vertical compatibility (library updates) as well as horizontal compatibility (a client provider role) first. Then we check how successfully is incompatibility detected by existing tools with concert to both source and binary compatibility.

II. Related Work

In the technical domain, the term compatibility denotes[1] 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 [3], [1], [10], [2] and technical [5], [9], [8] literature, mostly dealing with the issue of either correct replacement or interoperability.

...

Overview of possible API breaking changes is collected in [4]. ...

Belguidoum [1] distinguishes between *vertical* and *horizontal* compatibility which may be respectively paraphrased as backward and client-provider compatibility. Vertical compatibility is important for library vendors if they desire to produce new versions that are backward compatible with older ones and thus library update in a system will be smooth. On the other hand, horizontal compatibility is important for a system developer who checks if a system works with a selected set of libraries.

...

[1]Source: the Merriam-Webster dictionary.

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.

The Java Language Specification formally defines a notion of binary compatibility [6, ch. 13] that is strictly defined with respect to the static analysis performed during linking. It significantly differs from the notion of source compatibility which is checked by the compiler as the consistency between a program and a library.

What is more, binary compatibility does not implies source compatibility nor vice versa. For instance, specialising the return type of a method is source compatible as the compiler converts types, but not binary compatible as the linker expects exact types. On the other hand, changing generic type parameters is often binary compatible due to erasures, but not source compatible as the compiler checks generic parameter bounds.

Generally speaking, the reason for this discrepancy is in different rules enforced by the linker and the compiler. The discrepancy increases in time as a consequence of new features included into Java. Whereas the compiler is enriched with new type conversions, the linker still requires exact type matching. For instance, boxing and unboxing has been added in Java 1.5 to make working with primitive and wrapper types transparent for programmers, but the same conversion is not part of the linker. Hence, a primitive type is something different from its wrapper type for the linker.

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. In contrary, binary compatibility among the libraries is checked when the program is invoked. It may result in situations where a system may be compiled but cannot run or vice-versa.

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) by the client program. For instance, in Java, a method added to an interface is acceptable for the client invoking this interface, but represent an incompatible change for someone implementing that interface as all methods must be implemented.

Compatibility also includes the semantics of programs. Simply said, a program is compatible with its libraries if they behave as expected, consequently causing the whole system behaving as expected.

As handling the full compatibility from all directions is hard, simplified approaches are taken in practise. Most commonly, compatibility is checked on libraries API where a library is understood as compatible if the API is compatible with clients. In other words, it is expected that a library does not change (worsen) its behaviour as long as API is not incompatibly changed.

Although API may be enriched with sophisticated behaviour-related annotations such as pre/post conditions [7], practical applications still rely on syntactic API expressing type system, i.e. method signatures. It allows for automation and easy verification by tools. The only question, to be answered in this work, is how current tools cope with finding API breaking changes.

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 categories to handle complexity. The categories are: access modifiers, data types, exceptions, generics, inheritance, class members, other (non-access) modifiers and borderline (uncategorised) cases. Each category is filled with examples of API change with the desire to cover all possible changes. However, the data does not have to be complete as some cases did not have to be know in the time of building the corpus, or new cases may appear as the language evolve. To cope with this, 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 scenario where a client uses libraries, but the libraries in the corpus are simplistic and contain only API with dummy implementation. In real-word a more complex back-end would be invoked behind the API instead. 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 sub-directories, 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 changed element 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.

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 is provided in the form of source-codes 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
```

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

The script generate metadata by following steps, it:

1) read all changes stored in the `client`, `lib-v1`, `lib-v2` directories,
2) compile the client against the original library – it should succeed,
3) compile the client against the updated library – it may fail,
4) invoke the client compiled against the old library with the updated library – it may fail,
5) write result ("1/0") to the CSV file.

Step 3 and 4 fail when a respective change is source or binary incompatible. The scrip reads output of these steps and generates "1" or "0" to the CSV file depending of the success status of the steps.

To sum up, we make the corpus publicly available as a GitHub project hosted at:

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

Following sub-sections detail API changes separated into categories. Short discussions is provided to overview why a category is signification.

A. Data Types

Data types represent fundamental changes as they are used in the corpus for method and constructor signatures and field types. Changes of data types are also used in, generics, inheritance and exceptions and for this reason impacts other categories.

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

Explicit cases to cover primitive types were 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 due to simplifying 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. Once the linker decides correct conversions, it is 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

Exceptions are from the API evolution point of view interesting as they are all compile-time checked. Java distinguishes between so called checked and unchecked exceptions. The checked ones must be handled by a client code, either be propagated to upper levels or managed by the `try-catch` block. The unchecked exceptions are propagated automatically, but 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. In other words, 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 library is used in conjunction with an already compiled client, they will link smoothly together.

As the exception is a data type, a Java class, the corpus contains examples combining their possible changes for checked and unchecked variants. All combinations are binary compatible and their differ only in their impact on source compatibility.

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*. Simply said, generics are erased during the compilation from the caller site and their persist only on the target side.

Whereas a client code is verified by the compiler to correctly use generic types, a binary code that uses generics may be combined with the code not using generics. 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 binaries updated, the system will successfully link. Let us note, that such as system will likely fail later on a `ClassCastException`, but strictly from the compatibility point of view, such a change is compatible.

D. Inheritance

E. Members

F. Access Modifiers

G. Other Modifiers

H. Borderline Cases

V. Tools Challenge

We searched for tools that are capable of discovering API syntactic backward compatibility and included them into this work. 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 with incompatibility classification first, then we extracted only meta-data that represent incompatibility. 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 with incompatibilities and used string matching to detected changes in tools outputs. We collected the results in a CSV file with "1" meaning a correctly detected incompatibility and "0" meaning that a tool did not find the incompatibility.

The lines with compatible changes were removed from the meta-data to prevent false negatives. We had to analyse

Tool	Clirr	Japicmp	japiChecker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
Basin 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					
Usage									
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)

only provably incompatible changes because the current client does not handle all possible cases. An incompatible usage not expressed in the client may exist and if detected by a tool, it would not match with the expected compatibility leading to a wrong result.

The lines with 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 `!` (japicmp), `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
```

The new tools may be added to the benchmark simply by adding invocable lines to this script. The script can contain any bash invocable lines that lead to execution of a tool and must be able to 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 benchamrk 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 reveal that `clirr` may be a better choice in certain use-cases than other better performing tools.

`Clirr` is the worst performing tool, which is however 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` overall showing 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.

`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 as we wanted to detect how the tools deal with source compatibility.

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 for industry level usage.

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 observer usage and output of the tools. 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. A small drawback may be that it produces only a TXT output, which makes it less convenient for third-party integrations. 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 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] Carlos Canal, Ernesto Pimentel, and José M. Troya. Compatibility and inheritance in software architectures. *Science of Computer Programming*, 41(2):105–138, October 2001.

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

[5] 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.

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

[7] Barbara Liskov. Keynote address – data abstraction and hierarchy. *SIGPLAN Not.*, 23:17–34, January 1987.

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

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

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

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