Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

API Evolution and Compatibility: A Data Corpus and Tool Evaluation

Kamil Jezek[a] Jens Dietrich[b]

a. 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

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

Abstract The development of software components with independent release cycles is nowadays widely supported by multiple languages and frameworks. A critical feature of any such platform is to safeguard composition by ensuring backward compatibility of substituted components. In recent years, some tooling has been developed to help developers and DevOps engineers to establish whether components are backward compatible. We investigate the state of the art in this space by benchmarking such tools for Java. For this purpose, we have developed a compact benchmark data set of less than 200KB. Using this dataset, we study possible API changes of Java libraries, and whether the tools investigates can detect them. We find that only a small number of tools suitable to analyse API evolution exist. Those tools are only infrequently maintained by small communities. All tools investigated have shortcomings in that they fail to detect certain API incompatibilities.

1 Introduction

The popularity of platforms such as OSGi [35] and Maven [1] has created new challenges for software developers. In particular, components deployed as libraries (in Java, jar files) are pulled from repositories and applications are composed without compiling code against the very libraries used at runtime.

For instance, when the OSGi dynamic wiring mechanism is used, a bundle requiring a service is resolved against another bundle providing such a service, but the bundle is not necessarily the same that was used for compilation. There are two

Kamil Jezek, Jens Dietrich. *API Evolution and Compatibility:A Data Corpus and Tool Evaluation*.

In Journal of Object Technology, pages 1–24. Available at http://www.jot.fm/

abstractions that enable this: the use of version ranges in dependency declarations enables the use of a different version of a service, and the use of the OSGi service layer (such as OSGi declarative services) enables the use of another implementation of a service defined by a Java interface.

In Maven, a similar situation can arise due to the automation of transitive dependency resolution. Here, an application is compiled against the very libraries it is deployed with. However, if those libraries recursively depend on other libraries, it is still possible that other library versions are used than the ones used at compile time. Again, the use of version ranges in dependency declarations makes this possible.

The problem is that this bypasses some of the crucial quality assurance steps built into standard build and deployment processes, such as automated regression testing. A common argument made to solve (or often, to ignore) this problem is to reason about compatibility: if the correctness of an application was established (e.g., by means of regression testing) with respect to one component, then we infer correctness with respect to another component as we assume compatibility between both components. There are different reasons to make such assumptions. For instance, one might assume that all libraries providing a standardised service such as a JDBC connection used to interact with a relational database are compatible. The more common case however is that the two components are different versions of the same component, and in particular that a component is replaced by a later version, and in this case a developer might assume that the respective versions are compatible.

The question arises what compatibility means in this context. In general, compatibility is about preserving contracts between collaborating components and ensuring safe substitution. There is a vast amount of existing work on safe and correct composition of components, including ProCom [44], Sofa [7], or X-man [28].

The underling notion of contract has many facets [3], including classical API compatibility that can be expressed through the type system of the underlying programming language, but it also includes aspects like semantics, quality of service and licensing.

However, existing practical applications available to software engineers only cover the API aspect of the contract. Rama [41] defends this practice and claims that *"ideally, the users of a module need to look no further than its API"*. For instance, OSGi's components expose packages and services (Java interfaces) and binding is allowed if APIs match. No deeper *semantic* analysis is performed. The checks performed by Maven are even coarser. Maven composes components (JAR files) based on their symbolic versions. Once a referenced component exists in a repository, composition is allowed.

API-based compatibility checks simplify the issue from the developers point of view at the price of unsoundness: certain incompatibilities will be missed. This is a classical trade-off made between complexity and usability. The main rationale is that API compatibility can be investigated by means of static analysis that can be easily integrated into standard build and deployment processes. This is particularly beneficial as there are several static analysis tools that have now been widely adapted and are part of the standard toolbox used by many developers, such as `PMD`[1], `Checkstyle`[2], and `Findbugs`[3].

Recent empirical studies have clearly demonstrated the need for better tools: many

[1] https://github.com/pmd
[2] http://checkstyle.sourceforge.net/
[3] http://findbugs.sourceforge.net/

developers are not aware of the rules that specify when component evolution is compatible [14], and empirical studies have demonstrated that this causes issues in real-world systems [16, 40].

The aim of this paper is to review existing tools to check the compatibility of Java components as a starting point for future development and research. For this purpose, we collected several existing tools that can be used to check API compatibility, the complete list is shown in Table 2. The methodology used was to start with the analysis of some tools we were aware of, then adding tools referenced on the respective web sites, and finally considering further tools references on developer forums.

Our work addressed the following two research questions:

RQ1 Does any of the tools reliably check API syntactical compatibility?

RQ2 Does any of the tools correctly distinguish between source and binary compatibility?

This paper makes two contributions. Firstly, we catalogue existing API compatibility checking tools and investigate their capabilities. Secondly, we provide an extensible dataset used for benchmarking such (existing and future) tools.

The remainder of this paper is organised as follows: Sections 2 and 3 discuss related work and summarise some fundamental concepts related to compatibility. In Section 4, we discuss the dataset developed, followed by the evaluation of the various tools in Section 5. A brief conclusion is provided in Section 6.

2 Related Work

In the technical domain, the term compatibility denotes[4] 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 the research [8, 2, 48, 5] and the technical [19, 34, 33] literature, mostly dealing with the issue of correct replacement and interoperability.

2.1 API Evolution

Belguidoum and Dagnat [2] distinguish between *vertical* and *horizontal* compatibility. This can be paraphrased as backward vs client-provider compatibility. Vertical compatibility plays a role when vendors want to produce backward compatible libraries, which allow for smooth system updates. On the other hand, the purpose of horizontal compatibility is to aid the checking system composition. Using this terminology, our work targets vertical compatibility as we propose a data set simulating evolution of libraries and then use this to assess tools for checking API compatibility. However, we also study horizontal compatibility in order to build an oracle of compatible and incompatible changes. In particular, we have developed clients that invoke the API so that we could then check if an evolved API remains compatible with this client code. This provided us with the information about compatibility breaking change for every evolution.

Using this terminology, our work targets vertical compatibility as we propose a data set simulating evolution of libraries and then benchmark tools checking API

[4]Source: the Merriam-Webster dictionary.

breaking evolution. However, we also worked with horizontal compatibility to build oracle of compatible and incompatible changes. In brief, to detect if an API change is compatible, we had to develop a client code that invokes the API and then we could check if the new API remains compatible with the client code. This gave as information about compatibility breaking change for every evolution.

Both concepts should be taken into account in order to successfully produce and use components that are *"units of independent deployment and third-party composition"* [46, 4.1.1,].

Compatibility can be inferred from the verification of contracts between collaborating components. Beugnard et al [3] have pointed out that there are different types of contracts, including contracts that can be expressed via syntax-oriented APIs, semantics and quality of service. Even component meta data may be part of contracts that determines compatibility, consider for instance issues around the compatibility of open sources licenses [31]. While most component systems used in industry focus on the API aspect of compatibility, non-functional aspects have been investigated and considered by several authors [9], and led to several (often OSGi-based) implementations, including, Fractal [6], Sofa [36] and Treaty [15]. Semantic contracts can be expressed by using *pre-* and *post-conditions* in the tradition of Hoare Logic [23] and design by contract [30].

2.2 API Analysis

Several authors have investigated the evolution of Java APIs by means of static analysis, in many cases detecting cases of (horizontal and vertical) incompatibilities. This includes the work of Jezek et al [24, 25], Raemaekers et al [38, 39] and Ebad and Ahmed [18] on standard Java, and the work of Linares-Vasquez et al [29] on Android.

Rama and Kak [41] defend the focus on API compatibility checks and state that *"In this age of collaborative software development, the importance of usable APIs is well recognized"*. They propose several metrics that help to either design or recognise "good" APIs. API usability and design is also discussed by Myers and Stylos in [32]. Scheller [43] tries to automatically measure the usability of API in terms of interface complexity – complexity of methods, constructors, fields, etc. Sawant studied how APIs are used [42] and developed a meta-model of API usage. He also provided a parser to collect data from open-source systems and made collected data publicly available.

2.3 API Changes Categorisation

To analyse API evolution, it is important to understand which changes are contract-breaking. API-breaking changes for Java have been catalogued by des Rivières [13], this catalogue has directly influenced the design of the benchmark we have developed in order to assess and compare tools. The end user survey conducted by Dietrich et al [14] uses a similar catalogue.

2.4 Incompatibility Resolution

Cossette and Walker [10] have discussed several available techniques to refactor clients to adapt to changed APIs. They have also expressed the need for a data corpus to study API changes: "we need a collection of all points of breaking change between a set of API versions" and put together a set of five open-source libraries (Struts, Log4j,

jDOM, DBCP and SLF4J) to address this need. While using real-word programs has some obvious benefits, it remains incomplete as in particularly such a small set does not exhibit a complete set of possible changes, and is therefore unsuitable to benchmark tools. For this reason, we decided to create a synthetic dataset that exhibits a full set of compatibility issues while still being manageable in terms of size and complexity.

Raemaekers et al [37] have investigated the correlation between API breaking changes with several other properties such us number of modifications. Taneja et al [47] tried to automatically find changed methods replacements by employing metrics such as name similarity, method size and closeness of method arguments.

2.5 Mitigation Techniques

In our previous work we have demonstrated how changes to the Java compiler can mitigate certain binary compatibility problems that have been observed [26]. The solution proposed aims to simplify the situation by at least narrowing the gap between source and binary compatibility in Java. However, this requires some rather invasive changes to the Java compiler, and is therefore unlikely to be deployed into standard Java.

There are also proposals to address this problem on the model level, where checks performed on those models can guarantee compatibility. Such approaches include ProCom [44] and SaveCCM [22].

3 Background: About Compatibility

A senior JDK engineer once noticed that "every change is an incompatible change" (quote from [12]). I.e., every modification of a library may influence the way other libraries can use, interact, extend, observe or substitute it. Tools like compilers, linkers and static analysis tools define compatibility as API stability. That means that if a change in a library does not prevent clients from linking and/or compiling, the change is expected to be compatible, even if it results in changed behaviour or performance. For instance, while a change from `List` to `Set` is acceptable for assignment to a field typed to `Collection`, the change may have an impact on clients that rely on a particular order of elements in the collection.

The Java Language Specification formally defines acceptable API changes in terms of binary compatibility [20, ch. 13]: *"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 in order to establish 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, however, it is stated 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. Since both notions are not entirely consistent [13], situations where a system may

be compiled but cannot run or vice-versa may occur [16]. While binary and source compatibility are both used to describe types of compatibility that can be checked by means of static analysis at different times, behavioural compatibility [12] cannot be checked as easily, and it is therefore often only observed when programs are executed. Unit testing tools are often used in practice. The obvious limitation of testing is that (1) it is unsound, i.e. it cannot prove compatibility, only approximate it to some extent and (2) it is not available for checks in the context of runtime composition.

Compatibility also depends on how a library is used, for instance, whether a library is only used (i.e., its methods being invoked) or whether some of its types are being subtyped (used for extension in the sense of the object-oriented inheritance). For instance, a method added to an interface is acceptable for the clients invoking methods in this interface, but breaks source compatibility for existing subtypes.

4 A Data Corpus to Study API Changes

In order to conduct an evaluation of existing compatibility checkers, we needed a suitable data set. The data set we developed for this purpose consists of several small Java programs that all exhibit certain compatibility issues. We had considered the use of existing data sets, but none was suitable for our purpose. DaCapo [4] is rather small and does only contain one version for each program. The Qualitas Corpus [49] is larger and contains multiple versions for each program in its evolution edition, but the number of interesting evolution changes we wanted the tools to be expose to is very small relative to the overall size of the programs. We therefore decided to create our own synthetic data set. The data set proposed here contains small programs that model the evolution and usage of consecutive versions of a library.

4.1 Methodology

We organised the synthetic programs along three dimensions in order to ensure that the corpus is as complete as possible. Those dimensions are (1) *what* is changing, (2) *where* the change is applied and (3) *how* the respective code is changed.

We used ideas from section 13 of the Java Language Specification, the catalogue by Rivieres [13], work by Dietrich et al. [16] and information found on developer forums to guide the design of the dataset.

Firstly, we have defined eight categories to address the what dimension: access modifiers, data types, exceptions, generics, inheritance, class members, other (non-access) modifiers and miscellaneous.

Secondly, we have searched where those changes can appear and defined a set of seven Java language elements: class, inner class, interface, method, constructor, field, generic type.

Finally, we have investigated how a language element can be changed and defined a few possible change. Basically, a change could may be a removed element, an added element or a modified element. While the number of possible changes between elements differs, in most case we encounter four possible changes: one for addition, one for removal and two modifications: strengthening and weakening.

This results in 224 possible combinations (7 elements $\times$ 8 categories $\times$ 4 changes). However, the number of final test-cases differ due to following reasons: some combinations are not possible (e.g. interface methods may be only public, so no test for modified access modification is possible). We detected these situations simply

by trying all combination for passing Java compilation. On the other hand, certain combinations of an element and a category may contain more changes. This is the case for methods where two cases must be tested: a data type can be used either as a parameter or as a return type. Furthermore, the data type may have more then four modifications (for instance, also including boxing and unboxing for primitive / wrapper types), detailed later in Section 4.3). Finally, generics are divided into two more sub-categories containing generic wild-cards and generic parametrised types, producing another dimension. Although generics are relatively complex, capturing possible combinations for evolving API is fairly straightforward. Both wild-card and parametrised types may be parametrised by other types repeating the same modification patter such as addition, removal, etc.

We did not consider annotations as they serve as meta data that do not directly affect API compatibility and therefore we considered them as out-of-scope. We note however that annotations can be used to expose program semantics in APIs, for instance, when they are used to represent method pre- or postconditions [17].

The methodology we used focuses on the *complete coverage* of API evolution scenarios that can potentially break clients either during compile or run time. It contains 251 scenarios of API changes. The corpus is easy to extend due to its canonical structure, as described in the next section.

4.2 Structure

The corpus is split into three directories: `lib-v1`, `lib-v2` and `client`. As the names suggest, the directories contain a first (original base-line) version of a library, a second (evolved) version of this library and an executable (main) client application which uses the library. The directories model real-life scenarios. The respective libraries in the corpus are minimalistic on purpose. A triple consisting of the two versions of a library and a client program represents one API change and we refer to it as a *scenario*.

Each library as well as the client have sub-directories representing Java packages. The package names are constructed as follows:

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

In this representation, `category` is one of the eight categories, `element` is one of the seven applicable language elements and `change` describes the actual change as it was described above. For instance, a case named `dataTypeClassFieldBoxing` means that the type of a field defined in a class is changed from a primitive type to the respective wrapper type.

The corpus is provided in the form of source-code with an `ant` script to build the binaries. The script output is a set of 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>` triple described above):

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

The design based on a naming convention facilitates extensibility of the corpus by simply adding new cases to sub-directories (packages) following this convention. In detail, a user is expected to provide three Java source files to extend the corpus. One file is stored in the `lib-v1` directory to represent a library version, one file is stored in `lib-v2` to represent an update and finally one file represents a client code in the folder `client`. All the files should be in the same Java package (subdirectory) following the naming pattern `<category><element><change>`. In fact, the naming convention is not enforced, but recommended in order to facilitate work with the relatively large corpus.

The corpus also contains a simple script `compatibility.sh` to produce an oracle for tool evaluation. The scripts will compile and execute all scenarios, and will report compilation errors indicating source incompatibility, and linkage errors indicating binary incompatibility. The script generates a CSV file with three columns: the name of the scenario as described above, and two columns indicating source and binary compatibility – using "1" to indicate compatibility and "0" otherwise.

The script performs the following steps:

1. compile `lib-v1` and `lib-v2` directories
2. compile the client against `lib-v1.jar` – this is baseline and must succeed
3. compile the client against `lib-v2.jar` to check source compatibility – if this step fails with compilation error, a source incompatible change is detected
4. invokes the client originally compiled against `lib-v1.jar` with `lib-v2.jar` to check binary compatibility – if this step fails with linkage error, a binary incompatible change is detected
5. based on results from steps 3 and 4 append either 1 or 0 to the CSV file to indicate (in)compatibility

At the end of these steps a CSV file with information about source and binary compatibility for each scenario is generated. We use this file as an oracle to benchmark the tools. The oracle is basically produced by means of dynamic analysis (execution), and used to assess the tools that perform static analysis.

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

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

The repository also contains a pre-generated oracle that has been checked manually for consistency with the Java language specification [20]. This is to address the situation that the oracle can be generated by an implementation or version of Java not compatible with the specification.

The following sub sections contain a more detailed discussion of the corpus by category.

Table 1 shows the number of scenarios in each category and their typical impact on compatibility (source/binary). The following sections will discuss changes in each category in more detail, followed with examples to show a typical change in a category.

Category	Scenario	Incompatibilities
Data Types	49	s/b
Exception	26	s
Generics	88	s
Inheritance	16	s/b
Members	28	s/b
Access Modifiers	18	s/b
Other Modifiers	30	s/b
Miscellaneous	4	b

Table 1 – Corpus overview by category

4.3 Data Types

Many incompatibility problems are caused by changes to data types in the context of method, constructor and field signatures (descriptors), generics, inheritance and exceptions. We cover changes which occur in most object-oriented languages as well as changes specific to Java. The basic changes considered are:

- Del – a type is removed
- Inst – a type is added
- Gen – a type is generalised, for instance, `java.lang.Integer` is generalised to `java.lang.Number`
- Spe – a type is specialised, which is opposite of the previous case
- Mut – a type is mutated, a type is changed to an incompatible one that is neither a sub- nor a super type

To take some of the Java language-specific features, in particular the distinction between primitive and reference types, into account, we have added some additional change types to cover changes of primitive types:

- Narrow – a "specialising" conversion for primitive types, e.g. a change from `long` to `int`.
- Widen – the opposite of narrowing - a "generalising" conversion

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

- Box – a primitive type is converted to its wrapper type, for instance, `int` to `java.lang.Integer`
- Unbox – a wrapper type is converted to the matching primitive type

Changes in this category often result in subtle differences between source and binary compatibility. In particular, `Gen`, `Spe`, `Narrow`, `Widen`, `Box` and `Unbox` are conversions performed only by the Java compiler, not the linker. Therefore, these changes are always binary incompatible, but can be source compatible depending on usage. `Gen` is a usually source compatible conversion for a method parameter type

[5], while `Spe` is source compatible for a method return type due to Java's support for co-variant return types.

Changes in the categories `Del` and `Mut` are generally neither source nor binary compatible.

Examples of possible changes in method parameters follow:

```
# method parameter types: source compatible, binary incompatible
 Gen:    void method1(Integer param1) -> void method1(Number param1)
 Box:    void method1(int param1)     -> void method1(Integer param1)
 Unbox:  void method1(Integer param1) -> void method1(int param1)
 Widden: void method1(int param1)     -> void method1(double param1)

# method parameter types: source and binary incompatible
 Spe:    void method1(Number param1)  -> void method1(Integer param1)
 Mut:    void method1(Integer param1  -> void method1(String param1)
 Narrow: void method1(double param1)  -> void method1(int param1)

# method return types: source compatible, binary incompatible
 Narrow: double method1()  -> int method1()
 Box:    int method1()     -> Integer method1()
 Unbox:  Integer method1() -> int method1()

# method return types, source and binary incompatble
 Widen:  int method1()     -> double method1(double param1)
 Mut:    Integer method1() -> String method1(double param1)
```

4.4 Exceptions

Java distinguishes between checked and unchecked exceptions. Checked exceptions must be handled by client code, either by propagating them further or managing them using a `try-catch` construct. Unchecked exceptions are propagated automatically, but can be optionally caught as well.

The handling of exceptions in client code is checked only by the compiler, not the linker. As a consequence, changes to exceptions in method signatures only affect source but not binary compatibility.

When a library method is updated by adding a new checked exception, the original client code cannot be compiled and 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, it will successfully link. It is worth noting here that this can be misleading. For instance, while changing a library so that a method declares and throws a checked exception does not compromise linking, it is likely to have a profound effect on the behavioural compatibility: the client programs is likely to fail when the exception is actually thrown.

The corpus combines examples where exceptions in method signatures are added, removed, specialised, generalised or mutated, with variants for both checked and unchecked exceptions.

There is no binary incompatible example related to exceptions. Examples of source incompatible, but binary compatible changes include:

[5] There are some exceptions to this rule that occur if the compiler cannot resolve ambiguity between overloaded methods [20, sect. 15.12]

```
# Checked exceptions: Source incompatible, binary compatible
 Add: void method1()  ->  void method1() throws IOException
 Gen: void method1() throws FileNotFoundException
        -> void method1() throws IOException
 Mut: void method1() throws SQLException
        -> void method1() throws IOException

# Unchecked exceptions: source and binary compatible
 Add: void method1()  ->  void method1() throws NullPointerException
 Gen: void method1() throws NullPointerException
        -> void method1() throws RuntimeException
 Mut: void method1() throws NullPointerException
        -> void method1() throws IllegalArgumentException
```

4.5 Generics

Generics were added to Java relatively late in version 1.5 with strong consideration for compatibility with previous Java version. In order to achieve this, language designers opted for a design based on *erasures*. With erasure, type parameters are erased during the compilation from the call site. This means that during linking, only raw types without generics are checked. This is achieved by using descriptors (a non-generic version of the full generic signature) to describe method references at the call sites.

While client code is checked by the compiler for correct usage of generic types, binary code that uses generics may be combined with the code not using generics due to erasures. The impact on compatibility is evident. Changes that are binary compatible are not necessarily source compatible.

For instance, if a list is declared as `java.util.List<String>` only instances of `String` may be added to the list, and this is enforced by the compiler. However, when the definition is changed to `java.util.List<Number>` and only the binaries of the respective library are replaced, the program will successfully link. This is another case where (binary) compatibility is deceptive and issues are "shifted" into behavioural (in-)compatibility – the program is likely to fail at runtime with a `ClassCastException` as the compiler introduces `checkast` instructions that fail when the client program attempts to add strings to the list.

Examples:

```
# Parameterized types: Source incompatible but binary compatible
Mut: void method1(List<String> param1)  -> void method1(List<Integer> param1)
Gen: void method1(List<Integer> param1) -> void method1(List<Number> param1)
Spe: void method1(List<Number> param1)  -> void method1(List<Integer> param1)

# Wildcards: Source incompatible but binary compatible
Mut: void method1(List<? extends String> param1)
      -> void method1(List<? extends Integer> param1)
Spe: void method1(List<? extends Number> param1)
      -> void method1(List<? extends Integer> param1)

# Wildcards: Source and binary compatible
Gen: void method1(List<? extends Integer> param1)
      -> void method1(List<? extends Number> param1)
```

4.6 Inheritance

Some authors actively discourage implementations/extensions of types from APIs provided by libraries. For instance, Grand [21, p. 55] advises that: *"if a class is declared as a subclass, there is risk that these classes not under your control will change in an incompatible way"*.

Some changes to super types such as removed methods clearly break compatibility, but some breaking changes are less evident, such as the addition of an method to an interface or increasing the visibility of a method. For instance, changing the visibility of a method from `private` to `public` may seem harmless, but the new public method may overlap with the same method in some subtype. When the subtype method enforced stricter access, compilation fails as access cannot be weakened in overridden methods.

Some of those changes are covered by other categories (method, modifier, types etc. changes). This category contributes with several more examples with class/interface definitions modified in a subtypes. Several examples where methods are moved up and down the hierarchy tree are included as well. Examples in this category include:

```
# Access modifiers: source incompatible
Super class:  protected void method() -> public void method()
Sub class:    protected void method()

# Method introduced in interface: source incompatible
Interface:   X -> void method()
Sub class:   X

# Method removed: source incompatible when annotated with @Override
Interface: void method() -> X
Sub class: @Override public void method()
```

4.7 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 some impact on compatibility: removing members usually results in incompatibilities, but even adding members may be incompatible in the context of inheritance as discussed above.

Added abstract/interface methods with Java 1.8 `default` methods are modelled in the category as well.

4.8 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 compatible while restricting access is incompatible. This behaviour is consistent for source and binary compatibil-

ity. A special case is the increase of visibility in the context of inheritance as already discussed.

Example:

```
# Access decreased: Source and binary incompatible
 public void method() -> protected void method()
```

4.9 Other Modifiers

Other (non-access) modifiers have various purposes in Java and for this reason have different impacts to compatibility. The modifiers `volatile`, `transient`, `native` or `strictfp` signal special behaviour of the respective members, `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` is also used to denote classes that cannot be sub-classed, and methods that cannot be overridden.

There is no pattern how these modifiers impact compatibility. For instance, an added modifier `transient` does not break compatibility while adding `native` does. This is because `native` requires a special treatment by the JVM while `transient` is only meta-information. The modifiers `final` and `abstract` have the obvious effect that adding or removing them breaks the compatibility of inheriting classes.

An interesting case is the `static` modifier. The language permits the access to static fields and the invocation of static methods from non-static contexts, although most compilers emit a warning. However, changes (making a non-static method or field static or visa versa) are binary incompatible as different byte-code instructions are used for static and non-static access or invocation.

Examples:

```
# Static added: source compatible but binary incompatibe
 Method: void method() - > static void method()
 Field:  int field -> static int field

# Inheritance: source and binary incompatible
 Super class: void method() -> final void method()
 Sub class:   @Override void method()

# Inheritance: source and binary incompatible
 Super class: void method() -> abstract void method()
 Sub Class:   X (not inherited)

# Native: source compatible, binary incompatible
 void method() -> native void method()

# Transient: source and binary compatible
 void method() -> transient void method()
```

4.10 Miscellaneous

Java contains several specific features that are grouped in this category. It contains scenarios resulting from the implicit inheritance from `Object` by any classes, and the fact that any Java array implicitly implements `Cloneable` and `Serializable`.

Another example in this category is a change from a class to interface or vice-versa [27, Section 4]. This is interesting because there is no difference between the invocation of class and interface methods in source-code. However, different byte-code instructions are used, leading to binary compatibility problems that require recompilation of client code.

Examples:

```
# Interface to/from class: source compatible but binary incompatible
  class Foo <-> interface Foo

# Array type: source compatible but binary incompatible
 Gen: void method(String[] param1)  -> void method(Serializable param1)
 Gen: void method(String[] param1)  -> void method(Object param1)

 Spe: Serializable method() -> String[] method()
 Spe: Object method() -> String[] method()
```

5 Tool Evaluation

We have evaluated several tools that are available to developers in order to check the compatibility of API changes. The tools included are listed in Table 2 together with information about the authors, current versions, licensing and platform integration. All tools are basically static analysis tools that try to assess the compatibility of different versions of a program by creating models from (byte) code, and analysing those models.

5.1 Methodology

The selection of tools for the benchmark is driven by a single use-case. The user inputs two files – a version of a library and its update – and the tool produces a report listing API compatibility-breaking changes. Any tool supporting this use-case fits into this study. To find suitable tools we manually searched the Internet. We started with a few tools we knew about and searched for keywords such as “alternatives”, “replacement” etc. Some of the tools we found also refer to alternatives on their web-pages. Finally, we searched developer forums, most noticeably `stackoverflow.com`. The overall finding revealed that the number of existing tools is small and for this reason we included all of them.

The tools found were then evaluated using following approach: we first used the Java compiler and linker to create an oracle of incompatibility issues as described above in Section 4.

We then used the tools to be evaluated and captured the tool output in text files. Some tools also report compatibilities, we filtered this information out in order to avoid false positives. We used regular expressions for this purpose, tools flag positive

Tool	Clirr	Japicmp	japiChecker	JAPICC	Revapi	Sigtest	Japitools	Jour	JaCC
Basic info									
Author	Lars Kühne	Martin Mois	William Bernardet	Andrey Pono-marenko	Lukas Krejci	Oracle	Stuart Bal-lard	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	9/05	3/16	10/15	4/16	3/16	4/16	11/07	12/08	
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 2 – Tested Tools (GPL//LGPL = GNU GPL/LGPL, A2.0 = Apache 2.0)

output with text patterns that are easy to recognise, for instance `!` (`japicmp`), `100% Compatible` (`japitool`), `NON_BREAKING` (`revapi`) or `INFO` (`clirr`).

Finally, we compared the tool output with the oracle in order to establish which incompatibility issues were correctly reported by the tool.

5.2 Extendibility

The whole process is automated and may be invoked by a bash script `./benchmark.sh`. This script prepares the meta-data, invokes the tools, filters and formats the outputs and analyses results. The actual invocation of tools is delegated to script `tools/run.sh`, which executes all tools one-by-one.

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

```
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
```

Additional tools can be easily added to the benchmark by adding code to invoke the respective tool to this script. This script must ensure that the output of the respective tool is captured, formatted and stored in `tools/.reports`.

The structure of the corpus including the tools benchmark looks as follows:

```
<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
```

5.3 Results

The result of the experiment conducted indicate that the tools differ widely in their ability to detect compatibility-breaking changes. A result summary is provided in Table 3, in this table we report the percentages of successfully detected compatibility-breaking changes, classified by category.

While the results show clearly that the tool with weakest performance is `clirr` and the best is `sigtest`, detailed analysis reveals that `clirr` may still be a better choice than some of the better performing tools in certain circumstances.

Active development of `Clirr` stopped in 2005, and it is therefore not surprising that it does not recognize issues caused by the use of generics. Its focus is to check binary compatibility as defined in the Java Language Specification, and therefore it misses issues related to source compatibility, example is the Exceptions category. However, it works well in other categories and may be still useful for detecting only binary incompatible changes.

The situation is similar for `japicmp`. The results obtained are rather poor, however this is caused by a lack of support for generics and a few bugs in detecting modifiers. In all other categories, the tool performs well.

Another tool that generally performs well is `japitool`. Active development ceased in 2006, but the tool is still available as part of certain Linux distributions, including Debian.

Newer tools like `japicc` and `revapi` have a better overall score, but both have several issues scattered amongst several categories. They may be less reliable in production as they can miss some important source and binary compatibility issues. Nonetheless, both tools are still actively developed and may be therefore improved in the future.

`Sigtest` wins the benchmark as it is able to detect almost all problems. It fails to detect only two issues: (1) the removal of the `strictfp` modifier and (2) the addition of the `native` modifier, both causing incompatibility. We do not expect that those changes are very common in real-world programs.

Table 4 provides a more detailed analysis of results classified by whether the issue is source or binary incompatible. The first row shows changes that are source incompatible but binary compatible. The second row lists changes that are binary incompatible, but may be either source compatible or incompatible.

The table provides some interesting insights. In general, most tools perform much better in detecting binary incompatibilities. The exception is `revapi` which performs relatively poor here. On the other hand, most tools fall short in detecting source incompatibilities. The only tools that do so reliably are `sigtest` and `japitool`.

Other aspects like usability are also important properties to consider when selecting tools. A tool with a few bugs may be a better choice if it provides a better user experience. While we did not evaluate these aspects systematically in this work,

Category	clirr	jacc	japicc	japiChecker	japicmp
Access Modifiers	100.00%	100.00%	83.33%	100.00%	100.00%
Data Types	100.00%	100.00%	89.36%	100.00%	100.00%
Exceptions	0.00%	0.00%	100.00%	100.00%	100.00%
Generics	0.00%	33.33%	5.88%	0.00%	0.00%
Inheritance	71.43%	100.00%	71.43%	85.71%	100.00%
Members	100.00%	100.00%	84.21%	89.47%	100.00%
Other Modifiers	61.54%	84.62%	84.62%	53.85%	84.62%
Miscellaneous	100.00%	100.00%	75.00%	100.00%	100.00%
Total	57.79%	72.08%	59.74%	61.04%	65.58%

Category	japitool	jour	revapi	sigtest
Access Modifiers	100.00%	83.33%	83.33%	100.00%
Data Types	100.00%	100.00%	95.74%	100.00%
Exceptions 100.00%	100.00%	100.00%	71.43%	100.00%
Generics	100.00%	17.65%	100.00%	100.00%
Inheritance	100.00%	100.00%	42.86%	100.00%
Members	100.00%	84.21%	42.11%	100.00%
Other Modifiers	69.23%	76.92%	61.54%	84.62%
Miscellaneous	100.00%	100.00%	50.00%	100.00%
Total	97.40%	68.18%	82.47%	98.70%

Table 3 – Correctly Detected Incompatibilities in Each Scenario

we did make some observations. All tools provide a similar integration features and interfaces. For instance, all tools provide a command line interface (CLI) with options to input JAR files and produce a human readable formatted output. None of the formats used stands out. Only `japicc` provides HTML output which is useful to highlight the detected severity of changes, this could be better readable by users.

To summarise our findings, we answer the research questions as follows:

RQ1 – Does any of the tools reliably check API syntactical compatibility The answer is yes, the tools do exist but their ability varies. The recommended tool according to our evaluation is `sigtest`, which is distributed as open-source and may be easily integrated into the development process via CLI, Maven or Ant plugins. Another well-performing option is `japitool`, missing only a few incompatible modifiers, but the main issue is that the tool is not maintained. Other tools miss 17% or more of the incompatibility patterns identified and should therefore be used with caution

RQ2 – Does any of the tools correctly distinguish between source and binary compatibility The answer is yes and the most suitable tool is again `sigtest` but also `japitool`. Many alternative tools are still suitable if only binary compatibility checks are required. But this may be sufficient in many scenarios as library updates are usually distributed in binary form. Hence, binary compatibility checks may help to find issues that would otherwise result in runtime failures caused by opaque third-party libraries. Although a source incompatible change may break a system as well, it is easier to detect during builds of client programs early in the development process and therefore less harmful.

Type	clirr	jacc	japicc	japiChecker	japicmp
Source	13.24%	41.18%	25.00%	20.59%	25.00%
Binary	93.02%	96.51%	87.21%	93.02%	97.67%
Both	57.79%	72.08%	59.74%	61.04%	65.58%

Type	japitool	jour	revapi	sigtest
Source	100.00%	38.24%	88.24%	100.00%
Binary	95.35%	91.86%	77.91%	97.67%
Both	97.40%	68.18%	82.47%	98.70%

Table 4 – Correctly Detected Scenarios Divided into Source and Binary Incompatibilities

5.4 Tools and Community Size

An important aspect to evaluate the "business-readiness" of tools are the communities supporting them. To put this into perspective, we gathered some data on the tools evaluated here, and also on some widely used static code quality (smell-detection) tools: `PMD`, `checkstyle` and `Findbugs`. We found that the communities behind the compatibility checkers are small.

Existing work on the usability and impact of open source software has also considered other aspects such as project activity level, development team/community size [11], amount of development activity, input from the development community and user interest [45] as measures of success. We have followed their approach and also measured the size of projects in terms of number of commits, lines of code, developers (contributors) and frequency of commits. The numbers were obtained from `https://www.openhub.net/` on 16 September 2016. `openhub` collects project statistics from the respective source control systems. Not all of the tools investigated were tracked. In particular, we were not able to get data for `sigtest` as its subversion repository can not be parsed by `openhub` and we did not find an alternative source for comparable data.

We also searched the popular Q&A website `stackoverflow.com` to see how the respective projects were discussed. We tried to search by tag first, but this produced no results for most tools. For this reason we did a plain text search with tool names. We manually checked that the respective queries produced relevant results. We do not expect a lot of false positives here as the tool names such as "revapi" are unlikely to have homonyms used in this context. One exception was "jour" colliding with a French expression and we did not find any relevant questions for the `jour` tool.

The results are shown in Table 5. It is evident that the static code checkers – the first five tools – have a much bigger code-base, more contributors, commits and finally are more discussed in the community. These tools are also under active development.

5.5 Threats to Validity

The main possible threat of this paper is data completeness. If there were more API changes not covered here, where the tools perform differently, this could change the overall result. We have tried to mitigate this by composing data from several sources: our own experience, existing academic research, the Java specification and the catalogue by des Rivières. Moreover, the dataset is extensible and the experiment can be repeated with new data.

	PMD	Checkstyle	Findbugs	Jenkins	SonarQube
KLOC	175	142	286	1086	785
Contributors	32	113	47	1665	106
last commit	2mo	2mo	2mo	3mo	2mo
commits	8726	6000	15336	91995	25337
QA	3217	3603	4002	48166	11499

	Clirr	Japicmp	japiChecker	japicc	Revapi	Sigtest	Japitools	Jour	JaCC
KLOC	5	9	21	n/a	6	9	27		
Contributors	3	15	1	3	5	n/a	3	1	2
last commit	2y	2mo	5mo	n/a	4y	5y	6mo		
commits	427	525	113	62	796	n/a	153	152	1898
QA	32	8	25	3	30	26	9	n/a	0

Table 5 – Comparison of Code Style Checkers and API Compatibility Tools
QA – [stackoverflow.com]

6 Conclusion

In this paper, we have investigated how existing open-source tools cope with detecting incompatibilities in evolving APIs. We found that while the tools vary in performance and have only small and in some cases none supporting community, there are some highly usable and accurate tools available. The best performing tool was `Sigtest`.

We have also created and made public a benchmark data set for compatibility issues that occur during program evolution which can be used for other studies. This corpus is unique as it contains a oracle of all API-breaking changes we are aware of, and a script to produce this oracle that can also be used to assess future and alternative compilers and Java runtimes. The synthetic corpus we have created for this purpose is compact and minimalistic by design.

Possible future work is on extending the benchmark to cover more aspects of inheritance, and to add scenarios that describe other aspects of compatibility such as subtle behavioural changes.

Acknowledgment

This publication was supported by the project LO1506 of the Czech Ministry of Education, Youth and Sports.

The authors would like to thank Michal Bratner and Rudolf Augusta for their thorough preparation of test data, and for their support to find tools and documenting their usage.

References

[1] Apache Maven. `https://maven.apache.org/`, 2016. [Online; accessed 28-November-2016].

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

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

[4] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

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

[6] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software: Practice and Experience*, 36(11-12):1257–1284, 2006.

[7] Toms Bures, Petr Hnetynka, and Frantisek Plasil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications*, pages 40–48. IEEE Computer Society, 2006.

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

[9] Lawrence Chung and Julio Cesar Prado Leite. Conceptual modeling: Foundations and applications. chapter On Non-Functional Requirements in Software Engineering, pages 363–379. Springer-Verlag, Berlin, Heidelberg, 2009.

[10] Bradley E. Cossette and Robert J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 55:1–55:11, New York, NY, USA, 2012. ACM.

[11] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in free and open source software development: theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006.

[12] Joseph D. Darcy. Kinds of Compatibility: Source, Binary, and Behavioral. `https://blogs.oracle.com/darcy/entry/kinds_of_compatibility`, 2008. [Online; accessed 28-November-2016].

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

[14] J. Dietrich, K. Jezek, and P. Brada. What Java Developers Know About Compatibility, And Why This Matters. *Journal of ESE*, August 2014. submitted to second review.

[15] Jens Dietrich and Graham Jenson. Components, contracts and vocabularies-making dynamic component assemblies more predictable. *Journal of Object Technology*, 8(7):131–148, 2009.

[16] 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.

[17] Jens Dietrich, David J Pearce, Kamil Jezek, and Premek Brada. Contracts in the wild: A study of java programs. In *Proceedings of the 31st European Conference on Object-Oriented Programming (ECOOP'17)*, volume 74. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

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

[19] 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.

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

[21] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[22] H. Hansson, M. AAkerholm, I. Crnkovic, and M. Torngren. Saveccm - a component model for safety-critical real-time systems. In *Euromicro Conference, 2004. Proceedings. 30th*, pages 627–635, Aug 2004.

[23] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[24] 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.

[25] Kamil Jezek and Jens Dietrich. On the use of static analysis to safeguard recursive dependency resolution. In *SEAA 2014*, pages 166–173. IEEE Computer Society, 2014.

[26] 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.

[27] Kamil Jezek, Jens Dietrich, and Premek Brada. How java apis break - an empirical study. *Journal of IST*, 2015. submitted to second review.

[28] K.-K. Lau and C. Tran. X-MAN: An MDE tool for component-based system development. In *Proc. 38th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 158–165. IEEE, 2012.

[29] 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.

[30] Bertrand Meyer. Eiffel: A language and environment for software engineering. *JSS*, 8(3):199–246, June 1988.

[31] Jay Michaelson. There's no such thing as a free (software) lunch. *Queue*, 2(3):40, 2004.

[32] Brad A. Myers and Jeffrey Stylos. Improving api usability. *Commun. ACM*, 59(6):62–69, May 2016.

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

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

[35] The OSGi Alliance. *OSGi Service Platform Core Specification*, June 2011. Release 4, Version 4.3.

[36] Frantisek Plasil and Stanislav Visnovsky. Behavior protocols for software components. *IEEE transactions on Software Engineering*, 28(11):1056–1076, 2002.

[37] Steven Raemaekers, Gabriela F. Nane, Arie van Deursen, and Joost Visser. Testing principles, current practices, and effects of change localization. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 257–266, Piscataway, NJ, USA, 2013. IEEE Press.

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

[39] 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.

[40] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: a study of the maven repository. In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 215–224. IEEE, 2014.

[41] Girish Maskeri Rama and Avinash Kak. Some structural measures of api usability. *Softw. Pract. Exper.*, 45(1):75–110, January 2015.

[42] Anand Ashok Sawant and Alberto Bacchelli. A dataset for api usage. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 506–509, Piscataway, NJ, USA, 2015. IEEE Press.

[43] Thomas Scheller and Eva Khn. Automated measurement of {API} usability: The {API} concepts framework. *Information and Software Technology*, 61:145 – 162, 2015.

[44] Severine Sentilles, Petr Stepan, Jan Carlson, and Ivica Crnkovic. Integration of extra-functional properties in component models. In Iman Poernomo Christine Hofmeister, Grace A. Lewis, editor, *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*. Springer-Verlag Berlin, Heidelberg, June 2009.

[45] Katherine J. Stewart and Sanjay Gosain. The impact of ideology on effectiveness in open source software development teams. *MIS Q.*, 30(2):291–314, June 2006.

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

[47] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of api refactorings in libraries. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 377–380, New York, NY, USA, 2007. ACM.

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

[49] Ewan Tempero, Craig Anslow, Jens Dietrich, Ted Han, Jing Li, Markus Lumpe, Hayden Melton, and James Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference*, pages 336–345. IEEE, 2010.

About the authors

Kamil Jezek is a Postdoc at University of West Bohemia, Plzen, Czech Republic. His research areas include compatibility, program analysis and verification. He works on static reconstruction of API from Java byte-code and its correctness checking.

Email: kjezek@kiv.zcu.cz
URL: http://relisa.kiv.zcu.cz/

Jens Dietrich is an Associate Professor at Massey University in New Zealand. Jens research interests are in the areas of software componentry and evolution and static analysis.

Email: J.B.Dietrich@massey.ac.nz
URL: https://sites.google.com/site/jensdietrich/