On Testing the Source Compatibility in Java

Jan Hýbl
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic
hybljan2@fit.cvut.cz

Zdeněk Troníček
Faculty of Information Technology
Czech Technical University in Prague
Czech Republic
tronicek@fit.cvut.cz

Abstract

When software components evolve, they change interfaces, which may break backward compatibility. We present a tool that facilitates checking whether a new version of component is source compatible with a previous version. This tool figures out the component interface and generates the client code that uses the component interface to maximum extent. If the generated client compiles against the new component interface, those two versions are more or less compatible. The tool can be useful for API authors.

Categories and Subject Descriptors D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement

Keywords Software evolution, Java, API evolution, source compatibility

1. Introduction

When a change in application programming interface (API) causes that a client that compiled against the previous API version does not compile against the new API version, we say that the change breaks source compatibility [1] and call it the breaking change. For instance, if a class has the method

```
Class<?> loadClass(String codebase, String name);
```

and a new version of this class adds the method

```
Class<?> loadClass(URL codebase, String name);
```

the calls of the method with the first argument `null` do not compile against the new version because the call is ambiguous. We identified the following breaking changes:

1. The `final` modifier added to a class, a field, or a method
2. The `static` modifier added to or removed from a class, a field, or a method
3. The access modifier changed on a class, a field, or a method (only changes from `protected` to `public` and from `public` to `protected` are relevant; for instance, a change from `public` to `private` is considered as an API member removal)
4. A class added, deleted, renamed, or moved
5. The superclass of a class changed
6. The interfaces implemented by a class changed
7. A method added, deleted, renamed, or moved
8. The method parameters changed
9. The method return type changed
10. The exceptions thrown from a method changed
11. A field deleted, renamed, or moved
12. The field type changed
13. The annotation target changed
14. The annotation elements changed
15. The default value of an annotation element deleted
16. An enum value added, deleted, or renamed

We take into account all breaking changes but two for compatibility testing. We do not consider "a class added" as a breaking change because it happens very often (any new class may break a client) and the situation when a client is broken is rare. For the same reason we do not consider "an enum value added".

2. JASCC

In order to facilitate checking of source compatibility of two APIs, we implemented a tool called Java API Source Compatibility Checker (JASCC) [2]. The tool first figures out the component API by scanning either the source code or the bytecode. The source code scanner is based on javac and the bytecode scanner exploits reflection. The gathered information is then used to generate a code of a "complete" API user and a "complete" API extender. The complete API user of a tested class C calls every constructor and every method of C and accesses every field in C. The structure of the API user is as follows:

```
public class TestedClassUser {
```


SPLASH '13, October 26–31, 2013, Indianapolis, Indiana, USA.
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2508094

```
   // constructors tests
   // methods tests
   // fields tests
   // ancestors tests
}
```

In addition to common constructor and method calls with arguments of appropriate types, the API user also contains calls with null arguments, so that we are able to detect constructor and method overloading in a next API version. If a constructor or method throws exceptions, we catch them at the call site. For instance, if the `play` method throws `IOException` and `InterruptedException`, the generated API user contains the method as follows:

```
public void play(C instance, int speed) {
   try {
      instance.play(speed);
   } catch (IOException e) {
   } catch (InterruptedException e) {
   }
}
```

The complete API extender of a class D declares a subclass of D that overrides every protected and public method in D and accesses every protected and public field in D. The structure of the generated API extender is as follows:

```
public class TestedClassExtender extends
   TestedClass {
   // constructors tests
   // methods tests
   // fields tests
   // unimplemented methods
}
```

If the API class contains abstract methods, the extender implements them so that we are able to catch changes in them.

3. Evaluation

We evaluated the tool on five open source projects: Google Gson, Apache HttpClient, Apache PDFBox, Apache log4j, and Joda Time. For each project, we performed the steps as follows:

1. We downloaded the source codes of several versions of the project
2. For each version,

a) we used JASCC to generate the API clients

b) we compiled the API clients against the selected version

c) we compiled the API clients against the next version; this either succeeded (when there are no breaking changes) or failed

d) if the compilation failed, we blacklisted in the configuration file of JASCC all the API members that caused compilation failure and repeated steps c and d until compilation succeeded

e) we manually investigated the API changes found in previous steps and classified them as either refactorings or non-refactorings

The changes on existing API members are classified as refactorings based on assumption that the semantics of API members is stable. The changes that add new API members (New abstract method, Overloaded method) are classified as non-refactorings.

Table 1: The counts of breaking changes (#ver. is the number of investigated versions, Total is the number of all changes)

Project	#ver.	Refactorings	Total	%
Gson	9	53	55	96%
Apache HttpClient	3	5	7	71%
Apache PDFBox	9	135	153	88%
Apache log4j	13	169	174	97%
Joda Time	14	45	49	92%

4. Related Work

To the best of our knowledge, there is no tool similar to JASCC. Concerning the API evolution, we found several works. Dig and Johnson [3] presented a study on API evolution, which is similar to ours; however, their approach was different: they selected always two versions of five components, manually analyzed the API changes, and then double-checked the result by a tool and heuristics.

The studies on API evolution are motivated by effort to build a tool for automatic migration between two versions of a component. Henkel and Diwan [4] described the CatchUp! tool, which records refactorings when the component evolves and enables to replay them later. The same idea has been implemented in the Eclipse IDE [5].

5. Conclusion

The differences in the counts of breaking changes of investigated projects are large. On projects log4j and Joda Time we can see that the number of breaking changes does not depend on the size of the project. Either of them has between 60 and 70 kLOC and the counts of breaking changes are very different.

References

[1] https://blogs.oracle.com/darcy/entry/kinds_of_compatibility.

[2] Java API Source Compatibility Checker (JASCC). http://java.net/projects/jascc.

[3] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring, *Journal of Software Maintenance and Evolution: Research and Practice,* Vol. 18, Issue 2, 2006, pp. 83-107.

[4] J. Henkel and A. Diwan. CatchUp!: capturing and replaying refactorings to support API evolution, *International Conference on Software Engineering*, pp. 274–283, 2005.

[5] Eclipse IDE. http://www.eclipse.org

[6] J. Hýbl. *Code generator for testing of compatibility of API: Master's thesis.* Czech Technical University in Prague, Faculty of Information Technology, 2013.