UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Reproducible Wrapper for API Migration

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Malte Knauf

Erstgutachter: Prof. Dr. Ralf Lämmel
Institute for Computer Science

Zweitgutachter: Andrei Varanovich
Institute for Computer Science

Koblenz, im August 2011

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Abstract

API migration refers to the change of a used API to a different API in a program. A special case is called wrapper-based API migration. The API change is done without touching the program but the old API is reimplemented by means of the from now on used one. This so called wrapper has the interface of the reimplemented API but uses the implementation of the new one. This is an interesting approach because the wrapper can be used in each program, which uses the old API. To make this approach reproducible we study a ranking-based method for implementing a wrapper, where we always implement the method with the highest priority depending on failing test cases. Thus, we can reconstruct each implementation step.

We first develop an infrastructure to run and log test suites of Java projects that use an API, which we want to change. We then build a wrapper for a given API using the ranking-based approach.

Zusammenfassung

API Migration bezeichnet den Wechsel einer benutzten API eines Programms in eine andere. Eine spezielle Form ist die sogenannte wrapper-basierte API Migration. In diesem Fall wird die API gewechselt, ohne das eigentliche Programm zu berühren. Sie wird unter Benutzung der neuen API reimplementiert. Dieser so genannte Wrapper besitzt das Interface der alten API, benutzt jedoch die Implementierung der neuen. Dies ist ein interessanter Ansatz, da der Wrapper in jedem Programm eingesetzt werden kann, das bislang die alte API verwendet hat. Um diesen Vorgang reproduzierbar zu machen, entwickeln wir eine ranking-basierte Methode, um einen Wrapper zu entwickeln. Hierbei implementieren wir abhängig von fehlschlagenden Testfällen immer die API-Methode mit der jeweils höchsten Priorität. So können wir jeden Implementierungsschritt rekonstruieren.

Im ersten Schritt entwickeln wir eine Infrastruktur, mit der wir Testsuites von Java-Projekten starten und messen können. Mittels dieser Basis können wir nun die Ranking-Methode anweden, um einen Wrapper zu entwickeln.

Contents

1 Introduction

Nowadays there are many APIs that belong to the same domain. One can use any of them for the same task. The difference, however, may be the fact, that some feature of one API is implemented in another way than the others or it is missing at all. A problem can arise, if a program uses one API, which now cannot be used any more, either because the development of it is retired or because of the above mentioned implementation differences (e.g., he wants to use a feature that is missing or falsely implemented). The programmer must adapt his program to another API in the same domain, which will most likely affect large parts of his code because he must edit each existing API call and change it to the new API. This has to be done for every single program, that uses the old API. This change of the used API is called *API migration* and the above example is a very ineffective one.

1.1 Wrapper-based API Migration

There are several approaches to this issue. One way to do so is *wrapper-based API migration*. The interface of the old API is *wrapped* around the implementation of the new one. The actual program, that uses the old API, does not need to be touched because it still can call the old interface. In a previous study [1] Thiago Tonelli, Ralf Lämmel and collaborators analysed the wrapper-based approach and developed a wrapper for two given APIs.

The contribution of this work is to extend this approach in a way that it is reproducible. That is, the wrapper should be easy to implement, clearly understandable and easy to reconstruct with arbitrary API couples in the same domain. We take an API couple in the XML domain and develop a wrapper with it. In the next section we analyse an abstract method to gain reproducibility. In section 3 we present an infrastructure for building a reproducible wrapper based on the method of the previous section. In section 4 we apply our method and infrastructure to a specific pair of APIs and develop a wrapper for it.

2 Method

Our aim is a reproducible wrapper. To limit the work we cannot just simply reimplement a whole API. Also this would make it difficult to find a reproducible method for implementing the wrapper. Instead we take an existing program that uses the API we want to reimplement and we only implement API features that are used in the program. To do so, we need to log API calls in order to know which feature is used and which is not. If we furthermore specify in which order we implement these features (and document this order) everyone can retrace our implementation process and so our wrapper is reproducible. Figure 1 shows a component diagram of our infrastructure.

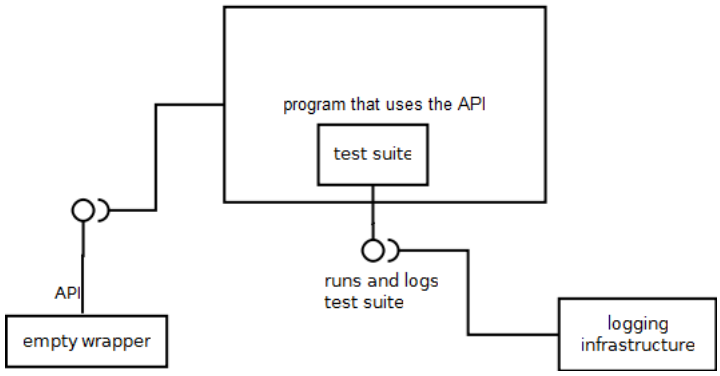

Figure 1: Component diagram of the infrastructure

As seen in the diagram we need to specify the following steps:

1. We need a program that makes use of the old API. This could be any open source project. As mentioned earlier we use this program to verify our wrapper.

2. This project needs a test suite which extensively uses this API. API features can either directly be called from a test case or indirectly from a program fragment which is tested within a test case. It does not need to use every feature of the API.

3. We start developing an empty wrapper. It should have the complete interface of the new API but instead of implemented bodies each call to this wrapper should raise a runtime error. That is, we can exchange it for the original API in the project of step one and be still able to compile it. However, every API calling test case should fail. This will be our the start of our implementing process.

4. We build an infrastructure to track each API call and count succeeding and failing test cases at runtime of the test suite. With this infrastructure it should be possible to see, which API features are called in which test case and if these tests fail or succeed. With these values we can analyse, which feature per test case is most likely responsible for failing this test case and set up a ranking that is sorted by frequency of these features. We hold the most recently called API feature responsible for each test case, because it is likely that otherwise many API features have the same ranking position. For example we run a test suite with an empty wrapper. Some tests cases call the same API features. These tests should fail directly after calling the first feature. The calls are counted and displayed in the ranking. When we implement the feature with the highest ranking position and run the test suite again, a new API feature is probably called afterwards, which will now lead to the

failing of the test case. Because the test case still fails, both API features are equally often called in the failing test cases and thus are on equal position in the ranking. To overcome this problem, we give priority to the recently called feature because it is the one we have not touched yet. Now we can implement the wrapper using the following algorithm shown in figure 2:

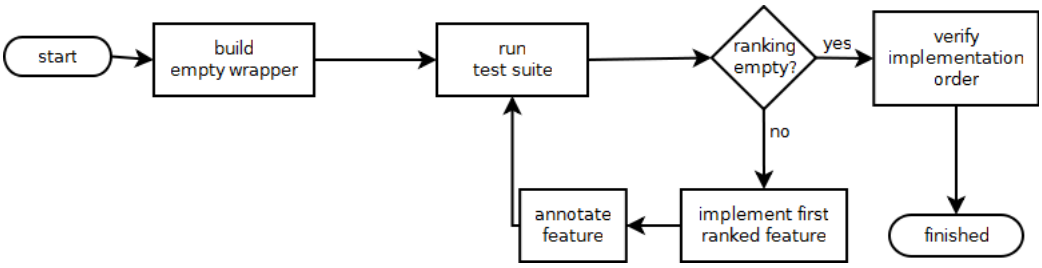

Figure 2: Flow chart of the implementing algorithm

(a) Run the project's test suite to get a ranking with the most likely failing API feature.

(b) Implement it.

(c) Go on from step 1 if the ranking is not empty.

After successfully applying the algorithm the test suite should succeed.

5. To prove the correctness of the ranking-based development, we need to know the implementation order of all implemented API features. We can achieve this by numbering each of them, which will represent the stage of development. The first feature will get position number 1, the second one position number 2 and so on. To prove the correct next implementation step of an arbitrary stage of development, we need to be able to fall back to this stage. This can be done by switching off each body of an API feature with a position number greater than the chosen stage. When we run the test suite after switching off certain API features the ranking of this stage of development should be shown. If the most likely failing API feature of this ranking is the one that was implemented next (according to its position number), we have proof that our ranking-based development was correct.

3 Infrastructure

In this section we will explain our developed infrastructure according to the requirements of section 2 that is needed to start our ranking-based wrapper development. We first choose our languages and tools to work with:

- Our programming language is *Java*

- Our test framework is *JUnit*
- Our build tool is *Apache Ant*
- Our analysing language and framework is *AspectJ*

As mentioned above we need a project (written in Java) with a test suite that makes use of the API we want to reimplement. We chose Ant because many Java projects are built with an Ant build script. So the build file of our infrastructure can comfortably call the project's build file. With the aspect oriented language AspectJ we count every failing and succeeding test case of the test suite and notice each API call. We also have functionality to switch API methods off. See subsection 3.2 for more detailed information on our use of AspectJ.

3.1 Ant Build File

This section describes the main build file of our infrastructure that is responsible for compiling and running everything related to this project, including our wrapper and the Java project, which uses the wrapper. See appendix A on how to run it.

- This script lets us choose to run the project's test suite either with its original implementation of the used API, with an empty wrapper or with the implemented wrapper. It simply copies the respective API jar into the directory, where the original API was located before. In case of the implemented wrapper it will be compiled before. Afterwards the project's ant file is called for compiling the project and its test suite.
- We cannot use the original ant target to run the JUnit test because we have to weave the aspects here and provide functionality to produce an output with the results of our measurements. We use an own target instead, which runs the test suite. We also use a master test suite, which calls the project's test suite(s). This is necessary because we need to produce an output at the end of the test run without touching existing test suites.

Each test case of the suite (or at least each test case that uses API methods) should succeed with the original API. If not, we have to ignore the failing tests to have a proper baseline. With the empty wrapper none of the API calling test cases should succeed.

3.2 AspectJ

We use AspectJ to log each API call in the running JUnit test suite and to log each test case run. We could have done that by explicitly writing the same logging code in each test case and API method but this would be very ineffective. Since this is a classical crosscutting concern and aspect oriented language like AspectJ provides

an obvious solution. With AspectJ for logging we don't need to touch existing code of the CDK and we don't need to know where API calls are made.

Because the source code of some methods that use API methods is not available (since some external jars may also use the API), we have to binary weave our aspects into the bytecode. We use *Load-Time Weaving* because otherwise we would have had to manually provide every single jar of the project to the weaver.

```
<target name="test">
    <junit printsummary="yes" haltonfailure="no" fork="yes" maxmemory="1500M">
      <jvmarg value="-javaagent:${lib.dir}/aspectjweaver.jar"/>
                ...
                <test name="main.CDKMastersuite"
                  haltonfailure="no"
                  todir="results"
                  outfile="result">
                <formatter type="brief" />
              </test>
  </junit>
</target>
```

Listing 1: Ant Target For Load-Time Weaving

In this Ant target (which is the one that starts the test suite) we provide the AspectJ-Weaver to the Java VM. We also provide an aop.xml to the weaver, which required for Load-Time Weaving. In this XML file we specify which packages and classes we want to weave.

With this base we count succeeding and failing test cases. These numbers are necessary for the ranking (see next section). We use a pointcut on each method that is annotated with a *@org.junit.Test* annotation, i.e., each Junit 4 test case.

- With a *before* advice we increment the total number of test cases.
- With an *after* advice we decide if the just advised test case succeeded or failed, based on a possibly thrown and a possibly expected exception. No thrown and no expected exception or thrown and expected exception are the same is interpreted as success; each other case is interpreted as failure.

3.3 Ranking

To set up a ranking of failing API methods, we need to store the name of each API call. To do so, we use another pointcut on each method and constructor within the API's package:

- With an *before* advice we store the signature's name of each API method, which is called within the same test case in an initially empty stack named *apiCallsPerTest*. The lastly called API method will be on top of the stack.

- After returning or throwing of the test case we add each called API method within the stack to a HashMap *overallFeatures*, which maps method signatures to a count of succeeding and failing calls. According to the result of the test case (succeeded or failed), the respective count of any called API method is incremented. The next test case will again start with an empty HashMap (see listing 2).

```
1  public static void updateFeatureMap(String testResult) {
2      ExecutionCount executions;
3      ...
4
5      for(String feature : apiCallsPerTest) {
6        if(!overallFeatures.containsKey(feature)) {
7          executions = new ExecutionCount();
8        } else {
9          executions = overallFeatures.get(feature);
10       }
11
12       executions.count++;
13       if(testResult.equals("succeeded")) executions.successes++;
14       else if(testResult.equals("failed")) executions.failures++;
15       else executions.errors++;
16
17       overallFeatures.put(feature, executions);
18     }
19
20     apiCallsPerTest.clear();
21   }
```

Listing 2: Ranking

3.4 Annotation

- In the ranking-based process each implemented method is annotated with an annotation *@Ranking(position=<value>)*. The value of the first implemented method is 1 and is incremented for each next method.

- The annotation *@Wrapping* is used for every auxiliary method and constructor within the API, which is not a public API method. These methods are not shown and counted in the ranking.

3.5 Verification

For the functionality to switch off certain API methods we also use AspectJ. We take a pointcut for API methods, which are annotated with the @Ranking-annotation and throw a runtime exception *¨not implemented yet¨* in a *before* advice, if the ranking number is equal to or greater than a given threshold.

```
1  after (): rankingMethods() {
2
3    if (Results.rankingThreshold != -1) {
4      String method = thisJoinPointStaticPart.getSignature().toString ();
5      for (int i = Results.rankingThreshold-1; i < Results.ranking.length; i++) {
6        if (Results.ranking[i].equals(method)) {
7          throw new UnsupportedOperationException("not yet implemented");
8        }
9      }
10   }
11 }
```

Listing 3: Verification Advice

We do not have reflection-based access to the position number within the @Ranking-annotation, because the heavy use of *thisJoinPoint* (which is required for reflection) uses too much memory in large test suites. Instead we manually provide every implemented method to an array *ranking* in which the first element represents the first implemented method and so on.

4 Study

The API pair of our study is *XOM*[1] and *DOM*[2]. We used the same Java project named CDK[3] as in [1] because it uses the XOM API and especially makes use of it in its test suite. Some test cases that use XOM methods fail with original XOM so we ignore them by annotating them with the JUnit 4 *@Ignore* annotation. We also slightly changed the Ant build script of the CDK in order to call it correctly with our own build script. See appendix B for the differences between the original and the changed files of the CDK.

4.1 DOM vs. XOM

In the following sections we first compare the two APIs in terms of metrics to get an overview of their complexity. We then present a mapping from DOM to XOM, that was required to develop the wrapper. After that we

4.1.1 Metrics

Table 1 and 2 show the numbers of public types and exceptions of each package of the XOM and DOM API. In the CDK only types of the *nu.xom* package are used. Table 3 shows the mapping of each used XOM type to the respective type of DOM. One can see that not every XOM type can be directly mapped to a type of DOM.

[1] XOM version 1.2.6, Link to the APIdocs: http://www.xom.nu/apidocs/

[2] w3c DOM. Part of Java SE 6, Link to the APIdocs: http://download.oracle.com/javase/6/docs/api/org/w3c/dom/package-summary.html

[3] Chemistry Development Kit, Link: cdk.sourceforge.net/

Package Name	#Types	#Exceptions
nu.xom	17	18
nu.xom.canonical	1	1
nu.xom.converters	2	0
nu.xom.xinclude	1	8
nu.xom.xslt	1	1

Table 1: XOM Packages

Package Name	#Types	#Exceptions
org.w3c.dom	27	1
org.w3c.dom.bootstrap	1	0
org.w3c.dom.events	7	1
org.w3c.dom.ls	10	1
nu.xom.xslt	1	1

Table 2: DOM Packages

XOM type	DOM type	#reimplemented XOM features
nu.xom.Attribute	org.w3c.dom.Attr	1/20
nu.xom.Builder	javax.xml.parsers.DocumentBuilder	3/15
nu.xom.Document	org.w3c.dom.Document	1/15
nu.xom.Element	org.w3c.dom.Element	12/38
nu.xom.Elements		2/2
nu.xom.Text	org.w3c.dom.Text	0/9

Table 3: Mapping of XOM types to DOM types

nu.xom.Elements, cannot be mapped to a type of DOM because there is no such type in DOM. Because it is simply a collection of Element types, it can easily be mapped to *java.util.ArrayList*. The third column of table 3 shows the number of XOM features (constructors and methods) per type that are used in the CDK and therefore reimplemented and the overall number of features.

XOM Ranking Table 4 shows the development order of the used XOM API methods and the ranking for the first five implemented methods.

For illustration we show the ranking of the first five implementations:

```
1 nu.xom.Builder() > count: 1412 | successes: 556 | failures : 24 | errors : 832
```

Because only one feature is avaiable in the ranking, we choose it.

```
1 Document nu.xom.Builder.build(Reader) > count: 703 | successes: 278 | failures: 12 | errors : 413
2 nu.xom.ParentNode() > count: 706 | successes: 278 | failures: 12 | errors : 416
3 Document nu.xom.Builder.build(InputStream) > count: 3 | successes: 0 | failures : 0 | errors : 3
4 nu.xom.Node() > count: 706 | successes: 278 | failures : 12 | errors : 416
```

Method or Constructor Name
nu.xom.Builder()
Document nu.xom.Builder.build(Reader)
Element nu.xom.Document.getRootElement()
String nu.xom.Element.getQualifiedName()
String nu.xom.Element.getBaseURI()
Elements nu.xom.Element.getChildElements()
int nu.xom.Elements.size()
Element nu.xom.Elements.get(int)
String nu.xom.Element.getNamespaceURI()
Attribute nu.xom.Element.getAttribute(String, String)
String nu.xom.Attribute.getValue()
Element nu.xom.Element.getFirstChildElement(String, String)
String nu.xom.Element.getLocalName()
String nu.xom.Element.getValue()
Elements nu.xom.ElementgetChildElements(String, String)
String nu.xom.Element.getAttributeValue(String)
Attribute nu.xom.Element.getAttribute(int)
String nu.xom.Element.toXML()
Document nu.xom.Builder.build(InputStream)

Table 4: Development Order of XOM Methods

ParentNode and *Node* are two abstract classes, so their constructor cannot be implemented and will be ignored in the ranking. *build(Reader)* will now implemented.

```
1 Element nu.xom.Document.getRootElement() > count: 703 | successes: 278 | failures: 12 | errors: 413
2 nu.xom.ParentNode() > count: 706 | successes: 278 | failures: 12 | errors: 416
3 Document nu.xom.Builder.build(InputStream) > count: 3 | successes: 0 | failures: 0 | errors: 3
4 nu.xom.Node() > count: 706 | successes: 278 | failures: 12 | errors: 416
```

The next method will be *getRootElement()*.

```
1 nu.xom.ParentNode() > count: 706 | successes: 278 | failures: 12 | errors: 416
2 Document nu.xom.Builder.build(InputStream) > count: 3 | successes: 0 | failures: 0 | errors: 3
3 String nu.xom.Element.getQualifiedName() > count: 703 | successes: 278 | failures: 12 | errors: 413
4 nu.xom.Node() > count: 706 | successes: 278 | failures: 12 | errors: 416
```

The next one is *getQualifiedName()*.

```
1 String nu.xom.Element.getBaseURI() > count: 703 | successes: 278 | failures: 12 | errors: 413
2 nu.xom.ParentNode() > count: 706 | successes: 278 | failures: 12 | errors: 416
3 Document nu.xom.Builder.build(InputStream) > count: 3 | successes: 0 | failures: 0 | errors: 3
4 nu.xom.Node() > count: 706 | successes: 278 | failures: 12 | errors: 416
```

And now *getBaseURI()*.

With this procedure every method will be implemented.

4.1.2 Differences

The main difference between **DOM** and **XOM** is the fact that XOM is constructor-based while DOM uses the factory pattern to construct objects. Because we reimplement the XOM API with DOM we can call a DOM factory inside a XOM method or constructor and just return or store the constructed object.

```
1  public Builder() {
2    factory = javax.xml.parsers.DocumentBuilderFactory.newInstance();
3    factory.setNamespaceAware(true);
4    try {
5      builder = factory.newDocumentBuilder();
6    }
7    catch (javax.xml.parsers.ParserConfigurationException e) {
8    }
9  }
```

Listing 4: Wrapper Constructor Builder

In listing 4, which is the XOM Builder constructor of the implemented wrapper, one can see that we first need a factory object, which then creates a DOM Builder object. Another difference between XOM and DOM is the distribution of the API classes. All XOM classes are located inside the *nu.xom* package whereas the DOM classes are located inside the *org.w3c.dom* package but depend on classes within different packages. Table 3 shows that *javax.xml.parsers* is used to reimplement XOM. In listing 4 some of these javax-Classes are used.

Most XOM reimplementations are pure delegations of the equivalent DOM methods or need just a few adaptations like in listing 5:

```
1 public final String getNamespaceURI() {
2    String namespaceURI = wElement.getNamespaceURI();
3    if (namespaceURI == null) {
4      return "";
5    } else {
6      return namespaceURI;
7    }
8  }
```

Listing 5: Wrapper Method getNamespaceURI

This is a getter for the namespace URI of a XOM element. If available the URI is simply returned by the equivalent DOM method. But if not it returns *null* whereas the XOM method should return an empty string. So we need to extend the delegation a bit.

5 Related Work

As mentioned earlier [1] analysed an API couple and made a wrapper for it. [15] extended the work by studying design patterns for wrapper-based API migration.

There are several alternative approaches to the concern of migrating and adapting software. One approach is called *twinning* [11], where changes of the code are specified as *mappings*, which can be applied to the program or the new API. Another API mapping analysis (this time from one language to another) is discussed in [16]. [10] is a graph-based approach to compare two given APIs (or to versions of the same API) and makes recommendations on how to adapt to the new API. [5] studies refactoring and refactoring-based migration tools in the context of *API evolution*, which is API migration to another version of the same API. Wrapping is also used for different purposes. [2] describes wrapper-based methods for migrating legacy software using Web Services. The interface that receives and sends messages will be wrapped to understand new protocols. [3], [7], [6] and [9] analyse and develop tools to automatize wrapping processes. [13] describes methods for reengineering program interfaces as a preliminary stage to wrapping them. Tracing in unit testing with aspect oriented languages is also used for related purposes in different domains. [14, 4] analyse ways to debug software by means of these two tools. A graph-based traceability approach is discussed in [12]. [8] focuses on developing and creating automated traceability tools.

A User Manual

A.1 Requirements

- Java SE 6 JDK must be installed
- Apache Ant 1.8 must be installed
- JUnit 4 must be available in Ant classpath
- aspectjrt.jar (AspectJ Runtime 1.6) must be available in Ant classpath

A.2 Running

The Ant build file to run the project is located in the project's root directory *dom-as-xom/build.xml*. In this directory run Ant with:

- **ant run_cdk_original** to run the CDK test suite with the original xom API.
- **ant run_cdk_wud** to run the CDK test suite with our wrapper in development (wud).
- **ant run_cdk_empty** to run the CDK test suite with the empty wrapper. That is, each API method will throw an exception.

To switch off each API methods in our wrapper, which is equal or greater than a given value add **-Dposition=<value>** between **ant** and the target, e.g., **ant -Dposition=10 run_cdk_wud**. This only works with the target **run_cdk_wud**.

Running the test suite can take up to 30 minutes. After finishing an output result.txt is created at *dom-as-xom/results*, which contains the measurements of AspectJ.

B Changed CDK Files

CDK/build.xml:

- Line 263 and 268: added *fork="true"* and *dir="."* attributes to java task

The following test cases have been ignored because they failed with the original XOM:

- org.openscience.cdk.io.cml.CMLRoundTripTest
  - testIsotope_ExactMass()
  - testIsotope_Abundance()
- org.openscience.cdk.io.cml.CML2Test
  - testSFBug1085912_1()
- org.openscience.cdk.qsar.descriptors.atomic.IPAtomicLearningDesciptorTest
  - testFluorobenzene()
- org.openscience.cdk.qsar.descriptors.atomic.PartialPiChargeDescriptorTest
  - testPartialPiChargeDescriptoCharge_2()
  - testPartialPiChargeDescriptoCharge_3()
- org.openscience.cdk.qsar.DescriptorEngineTest
  - testAvailableClass()

References

[1] Thiago Tonelli Bartolomei, Krzysztof Czarnecki, Ralf Lämmel, and Tijs van der Storm. Study of an API migration for two XML APIs. In *Postproceedings of Software Language Engineering (SLE 2009)*, LNCS. Springer, 2010.

[2] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *J. Syst. Softw.*, 81:463–480, April 2008.

[3] Chia-Chu Chiang. Automated software wrapping. In *Proceedings of the 45th annual southeast regional conference*, ACM-SE 45, pages 59–64, New York, NY, USA, 2007. ACM.

[4] Wouter De Borger, Bert Lagaisse, and Wouter Joosen. A generic and reflective debugging architecture to support runtime visibility and traceability of aspects. In *Proceedings of the 8th ACM international conference on Aspect-oriented software development*, AOSD '09, pages 173–184, New York, NY, USA, 2009. ACM.

[5] Danny Dig and Ralph Johnson. The role of refactorings in api evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Israel Gold and Uri Shani. Wrapping dce/osf client/server applications. In *Proceedings of the USENIX Applications Development Symposium Proceedings on USENIX Applications Development Symposium Proceedings*, pages 1–1, Berkeley, CA, USA, 1994. USENIX Association.

[7] M. Li, O. F. Rana, M. S. Shields, and D. W. Walker. A wrapper generator for wrapping high performance legacy codes as java/corba components. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[8] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Enabling automated traceability maintenance through the upkeep of traceability relations. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 174–189, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Ken Martin. Automated wrapping of a c++ class library into tcl. In *Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, pages 16–16, Berkeley, CA, USA, 1996. USENIX Association.

[10] Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson, Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. A graph-based approach to api usage adaptation. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 302–321, New York, NY, USA, 2010. ACM.

[11] Marius Nita and David Notkin. Using twinning to adapt programs to alternative apis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 205–214, New York, NY, USA, 2010. ACM.

[12] Hannes Schwarz, Jürgen Ebert, and Andreas Winter. Graph-based traceability: a comprehensive approach. *Softw. Syst. Model.*, 9:473–492, September 2010.

[13] Harry M. Sneed. Program interface reengineering for wrapping. In *Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, pages 206–, Washington, DC, USA, 1997. IEEE Computer Society.

[14] John Stamey and Bryan Saunders. Unit testing and debugging with aspects. *J. Comput. Small Coll.*, 20:47–55, May 2005.

[15] Thiago Tonelli, Krzysztof Czarnecki, and Ralf Lämmel. Swing to SWT and back: Patterns for API migration by wrapping. In *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*, pages 1–10. IEEE Computer Society, 2010.

[16] Hao Zhong, Suresh Thummalapenta, Tao Xie, Lu Zhang, and Qing Wang. Mining api mapping for language migration. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 195–204, New York, NY, USA, 2010. ACM.