Study of an API migration for two XML APIs

Thiago Tonelli Bartolomei[1], Krzysztof Czarnecki[1],
Ralf Lämmel[2], and Tijs van der Storm[3]

[1] Generative Software Development Lab
Department of Electrical and Computer Engineering
University of Waterloo, Canada
[2] Software Languages Team
Universität Koblenz-Landau, Germany
[3] Software Analysis and Transformation Team
Centrum Wiskunde & Informatica, The Netherlands

Abstract. API migration refers to adapting an application such that its dependence on a given API (the source API) is eliminated in favor of depending on an alternative API (the target API) with the source and target APIs serving the same domain. One may attempt to automate API migration by code transformation or wrapping of some sort. API migration is relatively well understood for the special case where source and target APIs are essentially different versions of the same API. API migration is much less understood for the general case where the two APIs have been developed more or less independently of each other. The present paper exercises a simple instance of the general case and develops engineering techniques towards the mastery of API migration. That is, we study wrapper-based migration between two prominent XML APIs for the Java platform. The migration follows an iterative and test-driven approach and allows us to identify, classify, and measure various differences between the studied APIs in a systematic way.

1 Introduction

APIs are both a blessing and a curse. They are a *blessing* because they enable domain-specific reuse. They are a *curse* because they lock our software into concrete APIs. Each API is quite specific, if not idiosyncratic, and accounts effectively for a form of 'software asbestos' [KLV05]. That is, it is difficult to adapt an application with regard to the APIs it uses. We use the term *API migration* for the kind of software adaptation where an application's dependence on a given API (the *source* API) is eliminated in favor of depending on an alternative API (the *target* API) with the source and target APIs serving the same domain.

API migration may be automated, in principle, by (i) some form of source- or byte-code *transformation* that directly replaces uses of the source API in the application by corresponding uses of the target API or (ii) some sort of *wrapping*, i.e., objects of the target *API's implementation* are wrapped as objects that comply with the source *API's interface*. In the former case, the dependence on the source API is eliminated entirely. In the latter case, the migrated application still depends on the source API but no longer on its original implementation.

Incentives for API migration

One incentive for API migration is to replace an *aged* (less usable, less powerful) API by a *modern* (more usable, more powerful) API. The modern API may in fact be a more recent version of the aged API, or both APIs may be different developments. For instance, a C# 3.0+ (or VB 9.0+) developer may be keen to replace the hard-to-use DOM API for XML programming by the state-of-the-art API 'LINQ to XML'. The above-mentioned transformation option is needed in this particular example; the wrapping option would not eradicate DOM style in the application code.

Another incentive is to replace an *in-house* or *project-specific* API by an API of greater scope. For instance, the code bases of several versions of SQL Server and Microsoft Word contain a number of 'clones' of APIs that had to be snapshotted at some point in time due to alignment conflicts between development and release schedules. As the 'live' APIs grow away from the snapshots, maintenance efforts are doubled (think of bug fixes). Hence one would want to migrate to the live APIs at some possible synchronization point—either by transformation or by wrapping. The latter option may be attractive if the application should be shielded against evolution of the live API.

Yet another incentive concerns the *reduction of API diversity* in a given project. For instance, consider a project that uses a number of XML APIs. Such diversity implies development costs (since developers need to master these different APIs). Also, it may imply performance costs (when XML trees need to be converted back and forth between the different object models of the APIs). Wrapping may mitigate the latter problem whereas transformation mitigates both problems.

There are yet more incentives. For instance, *API migration may also be triggered by license, copyright and standardization issues*. As an example, consider a project where the license cost of a particular API must be saved. If the license is restricted to the specific implementation, then a wrapper may be used to reimplement the API (possibly on top of another similar API), and ideally, the application's code will not be disturbed.

The 'difficulty scale' of API migration

Consider API evolution of the kind where the target API is a *backwards-compatible* upgrade of the source API. In this case, API migration boils down to the plain replacement of the API itself (e.g., its JAR in the case of Java projects); no code will be broken.

When an API evolves, one may want to *obsolete* some of its methods (or even entire types). If the removal of obsolete methods should be enforced, then API migration must replace calls to the obsoleted methods by suitable substitutes. In the case of obsoletion, the transformation option of API migration boils down to a kind of inlining [Per05]. The wrapping option would maintain the obsolete methods and implement them in terms of the 'thinner' API.

Now consider API evolution of the kind where the target API can be derived from the source API by *refactorings* that were accumulated on an ongoing basis or automatically inferred or manually devised after the fact. The refactorings immediately feed into the transformation option of API migration, whereby they are replayed on the application [HD05,TDX07]. The refactorings may also be used to generate adapter layers (wrappers) such that legacy applications may continue to use the source API's interface implemented in terms of the target API [ŞRGA08,DNMJ08].

Representing the evolution of an API as a proper refactoring may be hard or impossible, however. The available or conceivable refactoring operators may be insufficient. The involved adaptations may be too invasive, and they may violate semantics preservation in borderline situations in a hard to understand manner. Still, there may be a systematic way of co-adapting applications to match API evolution. For instance, there is work [PLHM08,BDH+09] that uses control-flow analysis, temporal logic-based matching, and rewriting in support of evolving Linux device drivers.

Ultimately, we may consider ***couples of APIs that have been developed more or less independently of each other***. Of course, the APIs still serve the same domain. Also, the APIs may agree, more or less, on features and the overall semantic model at some level of abstraction. The APIs will differ in many details however. We use the term *API mismatch* to refer to the resulting API migration challenge—akin to the impedance mismatch in object/relational/XML mapping [Amb06,Tho03,LM07]. Conceptually, an API migration can indeed be thought of as a mapping problem with transformation or wrapping as possible implementation strategies.

The 'risk' of API migration

The attempted transformations or wrappers for API migration may become prohibitively complex and expensive (say in terms of code size and development effort)—compared to, for example, the complexity and costs of reimplementing the source API from scratch. Hence, API migration must balance complexity, costs, and generality of the solution in a way that is driven by the actual needs of 'applications under migration'.

Vision

API migration for more or less independently developed APIs is a hard problem. Consider again the aforementioned API migration challenge of the .NET platform. The 'LINQ to XML' API is strategically meant to revamp the platform by drastically improving the productivity of XML programmers. Microsoft has all reason to help developers with the transition from DOM to 'LINQ to XML', but no tool support for API migration has ever been provided despite strong incentive. Our work is a call to arms for making complex API migrations more manageable and amenable to tool support.

Contributions

1. We compile a diverse list of differences between several APIs in the XML domain. This list should be instrumental in understanding the hardness of API migration and sketching benchmarks for technical solutions.
2. We describe a study on wrapper-based API migration for two prominent XML APIs of the Java platform. This migration is unique and scientifically relevant in so far that the various differences between the chosen APIs are identified, classified, and measured in a systematic way. The described process allows us to develop a reasonably compliant wrapper implementation in an incremental and test-driven manner.[1]

[1] We provide access to some generally useful parts of the study on the paper's website: `http://www.uni-koblenz.de/~laemmel/xomjdom/`

Limitations

We commit to the specifics of API migration by wrapping, without discussing several complications of wrapping and hardly any specifics of transformation-based migration. We commit to the specifics of XML, particular XML APIs, and Java. We only use one application to validate the wrapper at hand. Much more research and validation is needed to come up with a general process for API migration, including guarantees for the correctness of migrated applications. Nevertheless, we are confident that our insights and results are substantial enough to serve as a useful call to arms.

Road-map

§2 takes an inventory of illustrative API differences within the XML domain. §3 introduces the two XML APIs of the paper's study and limits the extent of the source API to what has been covered by the reported study on API migration. §4 develops a simple and systematic form of wrapper-based API migration. §5 discusses the compliance between source API and wrapper-based reimplementation, and it provides some engineering methods for understanding and improving compliance. §6 describes related work, and §7 concludes the paper.

2 Illustrative differences between XML APIs

We identify various differences between three major APIs for in-memory XML processing on the Java platform: DOM, JDOM and XOM. The list of differences is by no means exhaustive, but it clarifies that APIs may differ considerably with regard to sets of available features, interface and contracts for shared features, and design choices. API migration requires different techniques for the listed differences; we allude to those techniques in passing only.

In the following illustrations, we will be constructing, mutating and querying a simple XML tree for a (purchase) order such as this:

```
<order>
  <product>4711</product>
  <customer>1234</customer>
  <!-- ... further children elided ... -->
</order>
```

2.1 This-returning vs. void setters

Using the JDOM API, we can construct the XML tree for the order by a nested expression (following the nesting structure of the XML tree):

```
// JDOM -- nested construction by method chaining
Element order =
  new Element("order").
    addContent(new Element("product").
      addContent("4711")).
    addContent(new Element("customer").
      addContent("1234"));
```

This is possible because setters of the JDOM API, e.g., the `addContent` method, return **`this`**, and hence, one can engage in *method chaining*. Other XML APIs, e.g., XOM, use **`void`** setters instead, which rule out method chaining. As a result, the construction of nested XML trees has to be rendered as a sequence of statements. Here is the XOM counterpart for the above code.

```
// XOM -- sequential construction
Element order = new Element("order");
Element product = new Element("product");
product.appendChild("4711");
order.appendChild(product);
Element customer = new Element("customer");
customer.appendChild("1234");
order.appendChild(customer);
```

It is straightforward to transform XOM-based construction code to JDOM because **`this`**-returning methods can be used wherever otherwise equivalent **`void`** methods were used originally. In the inverse direction, the transformation would require a flattening phase—including the declaration of auxiliary variables. A wrapper with JDOM as the source API could easily mitigate XOM's lack of returning **`this`**.

2.2 Constructors vs. factory methods

The previous section illustrated that the XOM and JDOM APIs provide ordinary *constructor methods* for XML-node construction. Alternatively, XML-node construction may be based on *factory methods*. This is indeed the case for the DOM API. The document object serves as factory. Here is the DOM counterpart for the above code; it assumes that `doc` is bound to an instance of type `Document`.

```
// DOM -- sequential construction with factory methods
Element order = doc.createElement("order");
Element product = doc.createElement("product");
product.appendChild(doc.createTextNode("4711"));
order.appendChild(product);
Element customer = doc.createElement("customer");
customer.appendChild(doc.createTextNode("1234"));
order.appendChild(customer);
```

It is straightforward to transform factory-based code into constructor-based code because the extra object for the factory could be simply omitted in the constructor calls. In the inverse direction, the transformation would be challenged by the need to identify a suitable factory object as such. A wrapper could not reasonably map constructor calls to factory calls because the latter comprise an additional argument: the factory, i.e., the document.

2.3 Identity-based vs. position-based replacement

All XML APIs have slightly differing features for data manipulation (setters, replacement, removal, etc.). For instance, suppose we want to replace the product child of an order. The XOM API provides the `replaceChild` method that directly takes the old and the new product:

```
// XOM -- replace product of order
order.replaceChild(oldProduct, newProduct);
```

The JDOM API favors *index-based replacement*, and hence the above functionality has to be composed by first looking up the index of the old product, and then setting the content at this index to the new product. Thus:

```
// JDOM -- replace product of order
int index = order.indexOf(oldProduct);
order.setContent(index, newProduct);
```

It is not difficult to provide both styles of replacements with both APIs. (Hence, a wrapper can easily serve both directions of API migration.) However, if we expect a transformation to result in *idiomatic* code, then the direction of going from position-oriented to identity-oriented code is nontrivial because we would need to match multiple, possibly distant method calls simultaneously as opposed to single method calls.

2.4 Eager vs. lazy queries

Query execution returns some sort of collection that may differ—depending on the API—with regard to typing and the assumed style of iteration. Another issue is whether queries are *eager* or *lazy*. Consider the following XOM code that queries all children of a given order element and detaches (i.e., removes) them one-by-one in a loop:

```
// XOM -- detach all children of the order element
Elements es = order.getChildElements();
for (int i=0; i<es.size(); i++)
  es.get(i).detach();
```

The above XOM code is operational because XOM's queries are eager, and hence the query results are fully materialized before the corresponding collection can be processed. Here is the apparent JDOM counterpart:

```
// JDOM -- illegal detachment loop
for (Object k : order.getChildren())
  ((Element)k).detach();
```

Alas, the execution of this code will throw an exception because `getChildren` returns essentially a lazy iterator on the actual content list of `order`; changing that list invalidates the iterator. Hence, an operational JDOM counterpart must explicitly 'snapshot' the query result, say, in an extra object array as follows:

```
// JDOM -- detachment loop with up-front snapshot
Object[] es = order.getChildren().toArray();
for (Object k : es)
  ((Element)k).detach();
```

Arguably, this difference can be mitigated both by a transformation or in a wrapper. Of course, such semantic differences may go unnoticed for some time, and schemes of snapshotting may lead to noteworthy performance penalties.

2.5 Un-/availability of API capabilities

When XML is used as a model in an MVC/GUI application, then an event system is likely needed. For instance, the DOM API allows us to register event listeners with different kinds of events. The following code fragment registers a listener with the `order` element, which invokes its handler for any sort of node insertion:

```
// DOM -- register a listener for node insertion
((EventTarget)order).addEventListener(
  "DOMNodeInserted", // mutation type
  new EventListener() {
    public void handleEvent(Event evt) {
      // ... handle event ...
    } }, false);
```

Neither JDOM nor XOM provide an event system. More generally, we may face API couples where the target API misses some (nontrivial) capability of the source API. In some cases, the capability may be added by extension techniques (e.g., subclasses). In other cases, conservative extension techniques may be insufficient. For instance, the addition of an event system to an XML API would crosscut a considerable part of the API.

2.6 Less vs. more strict pre-conditions

Typically, XML APIs make an effort to quietly handle exceptional situations as long as well-formedness of XML trees is not jeopardized and no other blatant programming error would go unnoticed. Still the APIs differ as to where to draw the line. Consider the following JDOM code fragment, which attempts to remove the `product` child of `order` twice:

```
// JDOM -- exercise borderline case for node removal
order.removeContent(product); // properly removes.
order.removeContent(product); // quietly completes.
```

The above code will execute quietly because JDOM's pre-condition is weak here: it does not insist that the argument node must be in the container on which removal is performed. In contrast, the following XOM code throws an (unchecked) exception:

```
// XOM -- exercise borderline case for node removal
order.removeChild(product); // properly removes.
order.removeChild(product); // throws!
```

Such differences in pre-conditions (likewise for post-conditions) are challenging in API migration. If these differences are simply addressed by defensive programming techniques, then code bloat and inefficiency may be the result. In particular, in the case of the transformation option of API migration, it is not straightforward to produce idiomatic (concise) code.

3 The API couple of the study

The reported study on API migration concerns the XOM and JDOM APIs, with the goal of reimplementing XOM in terms of JDOM.[2] That is, JDOM is wrapped as XOM,

[2] We use the current versions of those APIs: XOM 1.2.1 and JDOM 1.1.

API package	#Types	#Throwable	NCLOC
nu.xom	50	18	15783
nu.xom.canonical	1	1	716
nu.xom.converters	2	0	606
nu.xom.xinclude	3	11	1070
nu.xom.xslt	6	1	550
	62	31	18725

API package	#Types	#Throwable	NCLOC
org.jdom	21	6	3802
org.jdom.adapters	8	0	416
org.jdom.filter	7	0	328
org.jdom.input	6	1	1088
org.jdom.output	7	0	1915
org.jdom.transform	3	1	418
org.jdom.xpath	2	0	238
	54	8	8205

Table 1. Packages of the XOM & JDOM APIs

meaning that types with the original XOM interfaces are implemented as *wrappers* with JDOM objects as *wrappees*. XOM and JDOM are two prominent XML APIs for the Java platform. They have been developed independently, say, by different software architects, in different code bases, and based on different design rationales.[3]

The main reason why our study considers migrating from XOM to JDOM, rather than v.v., is the availability of a comprehensive API test suite for XOM. Although wrapping an older API (JDOM) as a newer one (XOM) might appear counter-intuitive at first, such scenario is plausible in practice since migration drivers such as legal issues do not necessarily follow technical criteria.

In the sequel, we present some basic metrics and architectural details about the two APIs. We also describe the scope and some limitations of the migration and the available means for test-driven development.

3.1 API package structure

Table 1 lists XOM's and JDOM's packages. For each package, the second column gives the total number of declared types (i.e., classes and interfaces) except any descendants of `Throwable`. The third column is concerned with the latter, i.e., it gives the number of exception classes. The last column lists NCLOC ('Non-Comment Lines of Code') per package as an indication of the size (code complexity) of the packages and the APIs.

Let us look at XOM's packages first. The `nu.xom` package is XOM's core package (the core API). All the other packages cover specialized feature themes: canonical XML, DOM and SAX interoperability, XInclude support, and XSLT integration. Our study only covers the core API; we omit the discussion of all other themes (packages) in the present paper.

JDOM's core resides in the `org.jdom` package; it matches roughly the types and features of XOM's core, but we will discuss the correspondence more precisely below. The remaining packages cover, again, specialized feature themes: DOM interoperability, content filters for query functionality, advanced de-/serialization support, and XSLT and XPath integration.

3.2 Core API features

Table 2 lists all types of XOM's core and the corresponding JDOM types that were needed for XOM's reimplementation. XOM's core is mainly matched by JDOM's core,

[3] See `http://www.artima.com/intv/jdom.html` for background on the design rationales.

nu.xom	#Implementations
Attribute	20
Attribute.Type	4
Builder	15
Comment	9
DocType	18
Document	15
Element	38
Elements	2
Namespace	9
Node	8
NodeFactory	11
Nodes	8
ParentNode	8
ProcessingInstruction	11
Serializer	35
Text	9
XPathContext	5
Core Total	225

org.jdom	#Implementations
Attribute	29
CDATA	6
Comment	6
Content	9
Document	41
Element	76
JDOMFactory	25
Namespace	7
ProcessingInstruction	15
Text	12
input.SAXBuilder	39
output.XMLOutputter	47
Core Total	312

Table 2. Metrics on the core XOM/JDOM classes

but two additional types from the packages `org.jdom.input` and . . . . `output` are needed; c.f., the right-hand side of Table 2. This is mainly because de-/serialization is part of XOM's core, whereas JDOM has designated packages for these functions. We omit exception types as well as package-private types in the table entirely.

For each type (row), we show the number of methods that the type explicitly implements. This metric can be seen as a proxy for the effort needed in API migration. In our study, for example, each such implementation required roughly one corresponding method implementation in the wrapper.

In some situations, we may want to consider additional metrics, however. One such example is an interface complexity metric, defined as the number of methods a type understands (possibly including inherited or abstract methods). The inclusion of abstract methods is of particular interest to framework APIs, which may declare operations with no framework-provided implementations.

Yet other metrics could take into account the fact that polymorphic implementations of the source API may need to be migrated differently depending on the specific receiver type. For instance, a given method implementation of the source API may have different pre- and post-conditions for different receiver types. Also, a given method declaration of the source API may be implemented on a base type, whereas the target API's class hierarchy requires implementations on derived types. Such issues break the regularity of a wrapper's implementation. In the study, the impact of these issues was limited.

The #Implementations numbers of Table 2 give an idea of the feature complexity of the core API and the relative contribution of the different API types. It is immediately obvious that XOM has fewer methods than JDOM. In fact, JDOM is known to provide many 'convenience methods', which explains this difference. Interestingly, the NCLOC numbers of the core packages in Table 1 clarify that *XOM is substantially more complex than JDOM* (in terms of code size). This difference involves several factors—also in-

TestCase	#Tests	#Assertions
AttributeTest	38	137
AttributeTypeTest	3	70
BaseURITest	76	98
BuilderTest	152	364
CommentTest	17	52
DocTypeTest	46	103
DocumentTest	23	98
ElementTest	68	233
LeafNodeTest	3	2
NamespacesTest	53	110
NodeFactoryTest	43	95
NodesTest	10	33
ParentNodeTest	15	79
ProcessingInstructionTest	19	85
SerializerTest	135	194
TextTest	18	50
Total:	719	1803

The list of test classes maps roughly to the core API classes. There are 685 additional test cases for the omitted themes of the XOM API. The *TestCase*s are JUnit test classes with the shown number of test methods. Each test method tends to involve a small number of tests as evident from the number of assertions. Finally, we should mention that XOM also comes with a separate harness of basic benchmarks to test the speed and memory footprint of XOM programs. We have not used these benchmarks in any manner, but it would be interesting to systematically compare XOM's performance with the one of a wrapper-based reimplementation.

Table 3. Metrics on XOM's test suite

cluding incidental ones such as programming style. Most importantly, however, XOM is known to make a considerable effort to guarantee XML well-formedness. It pursues this goal by means of heavy checking, which directly affects the NCLOC metric.

3.3 XOM's test suite

The study uses test-driven development to push for compliance of the wrapper-based reimplementation of XOM with the original XOM API. We use the excellent XOM test suite to this end. JDOM's test suite does not have any role in this effort. Table 3 describes XOM's test suite in more detail.

4 Wrapper-based API migration

We will describe a simple and systematic form of wrapper-based API migration. In particular, we reimplement XOM in terms of JDOM. Hence, application code can be completely preserved because it may continue to depend on the interface of XOM.

4.1 API mapping

We begin a wrapper-based API migration by mapping each source type and method to a suitable target type and method. Such mapping requires domain knowledge; types and methods are compared at the level of domain concepts and their operations.

When mapping source types, we distinguish *regular vs. irregular types*. We say that a type is regular if it corresponds to a single target type; otherwise, the type is irregular. Indeed, some source types may need to be associated with multiple target types; yet other source types may lack a counterpart.

nu.xom	org.jdom	#regular methods	#irregular methods
Attribute	Attribute	23	5
Attribute.Type	java.lang.Integer	1	3
Builder	input.SAXBuilder	11	4
Comment	Comment	11	2
DocType	DocType	20	2
Document	Document	23	4
Element	Element	39	12
Elements	java.util.List	2	0
Node		0	2
NodeFactory	JDOMFactory	0	11
Nodes	java.util.List	8	1
ParentNode		0	0
ProcessingInstruction	ProcessingInstruction	16	1
Serializer	output.XMLOutputter	12	4
Text	Text; CDATA	11	2
XPathContext		0	5
		177	58

The table misses one core type; see Table 2 for the full list. That is, `Namespace` is omitted because it is only used by the original XOM implementation.

Table 4. Metrics on the XOM/JDOM mapping

When mapping source methods, again, we distinguish *regular vs. irregular methods*. We say that a method is regular if it corresponds to a single target method provided by (one of) the target type(s); otherwise, the method is irregular.

Table 4 summarizes the API mapping for the XOM/JDOM study. We obtained the mapping posteriori by inspecting the wrapper types and methods. 75% of all source methods provided by the wraper are regular. There are 4 irregular source types. For instance, JDOM does not provide a common base class like XOM's `Node`; some of its polymorphic methods have their counterparts implemented in multiple JDOM types instead. Please note that the number of source methods per type in Table 4 slightly deviates from Table 2 because the wrapper places some of the method implementations at different levels in the class hierarchy when compared to the original XOM implementation.

4.2 Wrapper implementation

We begin with an 'empty' reimplementation of the source API as follows. Each interface of the source API is reused as is by the reimplementation. Each class of the source API is reimplemented with the same interface, but with 'empty' (exception-throwing) method implementations. This empty reimplementation is compilable by construction, and any application of the API's original implementation remains compilable. Applications can be redirected to the new implementation by replacement of the API's JAR, by aspect-oriented programming, or by (manually) changing package references.

The next step is to turn the empty types into proper wrapper types. Here we systematically apply the design pattern for object adapters, where we implement the API mapping (c.f., §4.1) as follows. Each wrapper class (i.e., each class of the reimplementation of the source API) is set up, if possible, as an object adapter with an object of the

target API as the adaptee (also called the wrappee). For instance, the different `Element` types of XOM and JDOM would engage in a corresponding wrapper class as follows:

```
package nu.xom;
public class Element {
  private org.jdom.Element wrappee;
  // implement interface of wrapper in terms of wrappee
}
```

A few special cases should be mentioned in passing. First, abstract wrapper types may not need any wrappee type. Second, when we implement the wrapper class for a source type with multiple associated target types, the wrappee type might need to be an imprecise upper bound, such as `Object`, and methods may need to perform type dispatch (e.g., via **`instanceof`**) to invoke methods on the wrappee.

We speak of a *minor wrapping disorder* if a single wrappee object per wrapper object is fundamentally insufficient for reimplementation. This could happen, for example, if the source API intrinsically assumes a richer state than the target API. For instance, a reimplementation of DOM in terms of XOM or JDOM would need to maintain extra state in order to provide an event system; c.f., §2.5. Such disorders may be encountered late during implementation efforts, and they may trigger amendments of the API mapping; c.f., §4.1.

We speak of a *major wrapping disorder* if method invocations on the source API (handled by the wrapper) may need to be deferred or even rejected because there is yet state missing for the corresponding invocations on the target API. For instance, a reimplementation of XOM or JDOM in terms of DOM is challenging because XOM/JDOM's constructors are not implementable in terms of DOM's factory methods; c.f., §2.2.

The XOM/JDOM study involves only one minor wrapping disorder. The type `nu.xom.Serializer` receives a writer through a constructor argument, whereas the associated type `org.jdom.output.XMLOutputter` receives the writer through method calls. Hence, the XOM type must store the writer throughout.

4.3 Levels of adaptation

Ir-/regularity of a source method is based solely on the number of its associated target methods. There is a richer scale of *adaptation levels* that usefully classifies reimplemented methods, however. In the following, we define the different adaptation levels for a given source method *m*.

Adaptation level 1 *m is a regular method with m' as the associated target method. The reimplementation of m only performs basic delegation of m to m' on the wrappee (including wrapping and unwrapping). Argument positions may also be filled in by defaults. this-returning may be turned into void methods and v.v.; c.f., §2.1.*

Adaptation level 2 *Additional adaptations are involved in comparison to level 1. That is, arguments may be pre-processed (converted or checked); results may be post-processed (c.f., §2.4); exceptions may be translated; error codes may be converted into exceptions and v.v.; the delegation may also be conditional, subject to simple tests of the arguments; c.f., §2.6.*

nu.xom	1	2	3	4	other
Attribute	16	7	1	4	0
Attribute.Type	2	2	0	0	0
Builder	0	14	0	1	0
Comment	7	2	3	1	0
DocType	12	8	1	1	0
Document	15	8	2	2	0
Element	23	16	10	2	0
Elements	2	0	0	0	0
Node	0	0	0	2	0
NodeFactory	0	0	0	11	0
Nodes	6	0	3	0	0
ParentNode	0	0	0	0	0
ProcessingInstruction	14	1	2	0	0
Serializer	2	1	8	1	4
Text	4	7	1	1	0
XPathContext	4	0	0	1	0
	107	66	31	27	4

Basic delegation (level 1) suffices for a bit less than half of all methods; more than a quarter requires some pre-/post-processing (level 2); the remainder needs to be composed from other methods (level 3) or developed from scratch (level 4). It turns out, however, that all level 4 methods were not at all complex and could be implemented without problems. There are a few methods of the `Serializer` class that are not associated with an adaptation level. These methods were not implemented because there was no straightforward way of doing so, and the sample application used in the study did not exercise these methods.

Table 5. Adaptations per level for XOM/JDOM

Adaptation level 3 *m is an irregular method. Its implementation may invoke any number of target methods, but without reimplementing any functionality of the target API. In informal terms, a level 3 method is one that is effectively missing in the target API but which can be recomposed from other methods of the target API.*

Adaptation level 4 *The level 3 condition of 'not reimplementing any methods of the target API' must be violated. In informal terms, level 4 methods violate the 'intention of reuse' for reimplementing the source API in terms of the target API.*

Table 5 shows the methods per type and adaptation level for the study. We have assigned these levels manually (by categorizing the implementation) and recorded them through method annotations on the wrapper types. The shown numbers depend on a 'judgement call' for the required compliance of the wrapper as discussed in the next section. ***The more one pushes for full compliance, the more methods would be pushed upwards on the level scale; also, the more complex some method implementations would get.*** We would like to generally avoid method implementations at the adaptation level 4. That is, any substantial violation of the 'intention of reusing' the target API runs fundamentally counter the motivation of API migration. Likewise, we would like to avoid complicated or inefficient method implementations at the adaptation levels 2–3.

5 API compliance

In simple terms, the wrapper-based reimplementation of the source API should be 'fully compliant' with the original (implementation of the) source API. Compliance could be interpreted in the sense of contract-based equivalence for the original implementation and the wrapper. In practice, APIs often lack comprehensive contracts (pre-/post-conditions and invariants). Hence, test-based methods are needed. Using such test-based methods, 'compliance issues' are gradually discovered, and possibly resolved.

In the following, we clarify the process for discovering compliance issues; we categorize these issues; and we defend the idea that some issues may remain unresolved. The XOM/JDOM study continues to serve as the running example.

5.1 Test suite-based compliance

A strong test suite for the source API appears to be a reasonable tool in establishing compliance of the original API and the wrapper-based reimplementation. However, an important insight of our work is that it may be prohibitively expensive to achieve full compliance with regard to such a test suite (because it may approximate contract-based compliance at a very detailed, idiosyncratic level). Indeed, in the study, we have ultimately accepted partial compliance with approx. 40 % of all test cases not producing the expected result with the wrapper:

- # XOM test suite – *all* test cases: 697
- # XOM test suite – *compliant* test cases: 417
- # XOM test suite – *non-compliant* test cases: 280

In general, a strong test suite for the source API may be the *initial driver* in pushing the wrapper towards some basic compliance. Such a test suite is even more useful if it clearly identifies mainstream API-usage scenarios that must not be disturbed by non-compliance. To limit effort, one would initially concentrate on a smaller core API and important API-usage scenarios, indeed.

In the study, initially, we used a considerably smaller core of XOM. For instance, we left out `Serializer` because XOM has already a serialization capability through its `toXml` method. Also, we left out `DocType` (i.e., DTD) support because it seemed difficult to provide such support in the view of JDOM's lack of comprehensive `DocType` support.

Ultimately, API migration is driven by the actual 'application under migration'. The application may call for an extension of the initially covered API and for the inclusion of more API-usage scenarios. In the study, we picked an application under migration by searching the SourceForge repository for an application that both makes substantial use of XOM and references XOM in (say, JUnit-based) test cases. The best fit was CDK.[4]

In general, one needs to push the wrapper towards full compliance with the application's test suite—potentially balancing the wrapper development effort and the degree of automation of migration. ***In the study, we reached full compliance without any need for manual adaptations of the application*** except for 3 test cases whose dependence on the order of XML attributes had to be relaxed. The following numbers only cover CDK's test cases that use XOM.

- # CDK test suite – *all* test cases: 752
- # CDK test suite – *compliant* test cases: 752
- # CDK test suite – *non-compliant* test cases: **0**

[4] Chemistry Development Kit (CDK) is a Java library for structural chemo- and bioinformatics; c.f., `http://sourceforge.net/apps/mediawiki/cdk/`. The used checkout of CDK does not pass all of its test suite even with the original XOM implementation. We have only looked into compliance for test cases that passed with the original XOM implementation.

nu.xom	#always	#sometimes	#never	#unused
Attribute	13 / 3 [_,11]	4 [**1**, 3]	-	11 / 25
Attribute.Type	3 [_,3]	-	-	1 / 4
Builder	1 / 2 [_,1]	7 [**2**, 5]	-	7 / 13
Comment	7 [_,7]	2 [**0**, 2]	-	4 / 13
DocType	8 [_,8]	5 [**0**, 5]	-	9 / 22
Document	7 / 1 [_,7]	12 [**1**, 11]	-	8 / 26
Element	15 / 21 [_,9]	28 [**13**, 15]	-	8 / 30 **[2,_]**
Elements	0 / 2 [_,0]	2 [**2**, 0]	-	-
Node	-	-	-	2 / 2
NodeFactory	-	4 [**0**, 4]	1 [0, 1]	6 / 11
Nodes	2 / 2 [_,2]	3 [**2**, 1]	-	4 / 7
ParentNode	-	-	-	-
ProcessingInstruction	9 [_,9]	1 [**0**, 1]	-	7 / 17
Serializer	3 / 3 [_,3]	8 [**3**, 5]	3 [0, 3]	2 / 13
Text	7 [_,7]	1 [**0**, 1]	-	5 / 13
XPathContext	0 / 1 [_,0]	-	-	5 / 4 **[1,_]**
Total	75 / 35 [_,67]	77 [**24**, 53]	4 [0, 4]	79 / 200 **[3,_]**

XOM/CDK: The first number in each cell shows the compliance level for XOM's test suite. The number after the slash (if any) shows the compliance level for CDK's test suite. Note that all CDK test cases succeed; hence there are no methods at levels #sometimes or #never.

[moves to #always, moves to #unused]: The numbers in square brackets (if any) describe the moves between the levels with the 'initial' position defined by XOM's test suite and the 'final' position defined by CDK's test suite. For example, Attribute had 11 methods moved from #always to #unused, 1 from #sometimes to #always, and 3 from #sometimes to #unused.

Table 6. Compliance levels in the XOM/JDOM study

One of the reasons of compliance with the application's test suite vs. non-compliance with the API's test suite is of course that any given application will exercise the source API only in a limited manner. However, this may be even true for a reasonable test suite of an API. Consider the following numbers that we determined in the study:

- # all implementations of the wrapper: 277
- # XOM test suite – exercised method implementations: 156
- # CDK test suite – exercised method implementations: 35

Hence, about 3/5 of all method implementations where exercised by the API's test suite, and only about 1/10 were exercised by the application' test suite. Inspection reveals that the API's test suite specifically misses many of the more trivial methods (such as getters and setters and diversely overloaded constructors).

5.2 Compliance levels

It is now a central question whether or not the application runs into any of the compliance issues manifested by the API's test suite. The following method can be applied in this context. Each API method can be associated with a *compliance level* relative to any test suite as follows:

- *always*: it is exercised in compliant test cases only.
- *sometimes*: it is exercised in both compliant and non-compliant test cases.
- *never*: it was exercised but never in compliant test cases.
- *unused*: it is not exercised at all in any test cases.

The status of each method with regard to the application's test suite can now be compared with its status with regard to the API's test suite. This comparison is visualized for the study in Table 6. ***The table illustrates that several methods with compliance issues with regard to the API's test suite are used without problems in the application.***

Type	Methods	Issue type	Domain	Status	Comment
Attribute	toXML()	Post	Serialization	resolved	JDOM's escaping is different from XOM's
Attribute	Attribute(String,String)	Pre		resolved	XOM allows colonized names in the first argument whereas JDOM does not
Element	detach()	Invariant		resolved	A root element must always remain attached.
Element	addAttribute(Attribute)	Throws		resolved	XOM throws MultipleParentException if argument is parented whereas JDOM throws IllegalAddException
Element	setBaseURI(String)	Pre	BaseURI	unresolved	XOM agressively checks URI for well-formedness and throws accordingly
Element	getBaseURI()	Post	BaseURI	unresolved	In XOM the result is absolutized and converted from IRI to URI if needed

Table 7. Samples of compliance issues in the XOM/JDOM study

Incidentally, there are even implementations that were not exercised by the API's test suite but are exercised (and found compliant) by the application's test suite. (See the numbers in bold face in the table for both of these effects.)

5.3 Discovery of compliance issues

In the test-driven process of pushing the wrapper towards compliance, one could simply focus on the *number* of compliant test cases. However, such plain focus would provide little insight into the underlying causes for failing test cases and the actual API mismatch. Also, it would provide no guidance with regard to the prioritization of non-compliant test cases. Instead, test-driven development is to be refined such that non-compliant test cases are incrementally examined and some API method is to be 'blamed' to have a *compliance issue*.

Table 7 shows a few samples of documented compliance issues in the study. The format of these entries will be clarified gradually. All discovered issues are recorded by means of method annotations on the wrapper types.

As an issue is discovered, a decision must be made whether or not effort is to be spent (immediately) on its resolution. If the issue was discovered through an ambitious test suite for an API, then it may be reasonable to refuse resolution—because the issue is considered either a) less relevant for actual applications, or b) too complicated for an automated approach, calling for a case-by-case migration instead. Table 8 summarizes all resolved and unresolved issues in the study. This relatively small number of issues was indeed discovered incrementally, and about half of the issues remained unresolved, while the 'application under migration' is still fully compliant.

5.4 Generic compliance issues

Compliance issues can be caused by differences in pre-/post-conditions, invariants, and throwing behavior. We call these issues *generic* in the sense that they are meaningful for APIs of any domain. The following definitions assume two APIs α and α' with identical interface. In the wrapping context, α is the original implementation of the source API, whereas α' is the wrapper (at a given stage of development).

(a) #resolved

Type	#Pre	#Post	#Inv	#Throws
Attribute	3	1	-	4
Attribute.Type	-	-	-	-
Builder	-	-	-	-
Comment	2	-	-	-
DocType	-	-	1	-
Document	6	-	-	4
Element	5	-	1	8
Elements	-	-	-	-
Node	-	-	-	-
NodeFactory	-	-	-	-
Nodes	-	-	-	-
ParentNode	-	-	-	-
ProcessingInstruction	-	-	-	-
Serializer	-	-	-	-
Text	-	-	-	2
XPathContext	-	-	-	-
	16	1	2	18

(b) #unresolved

Type	#Pre	#Post	#Inv	#Throws
Attribute	-	-	-	-
Attribute.Type	-	-	-	-
Builder	5	1	-	7
Comment	-	1	-	-
DocType	7	1	-	1
Document	-	1	-	-
Element	3	4	-	-
Elements	-	-	-	-
Node	-	-	-	-
NodeFactory	-	-	1	-
Nodes	-	-	-	-
ParentNode	-	-	-	-
ProcessingInstruction	-	-	-	-
Serializer	-	-	-	-
Text	-	1	-	-
XPathContext	-	1	-	-
	15	10	1	8

Table 8. Number of resolved and unresolved XOM/JDOM issues

We say that method *m* has a *PRE* issue if its pre-condition is stronger in α' than in α. If we think of α' as the intended replacement of α, then such an issue violates design-by-contract rules. The opposite situation also needs to be considered: we also say that *m* has a *PRE* issue if its pre-condition is weaker in α' than in α. In this case, no violation of design-by-contract rules is present, but α' is more (too) permissive than α.

In the latter case, the issue can be addressed by adding extra checked assertions to the too permissive implementation. In the former case, a more complex implementation may be needed. Table 7 shows two examples of *PRE* issues in the study. In fact, the one on `Attribute` is about a too strong pre-condition (because JDOM rejects colonized names where XOM does not); the one on `Element` is about a too weak pre-condition (because JDOM checks less for well-formedness than XOM). As it is clear from the table, one of the issues was not resolved—well-formedness checking is particularly difficult to add to JDOM without leading to code bloat and possibly adaption level 4.

Likewise, we say that *m* has a *POST* issue if its post-condition in α' is weaker than the one in α. Further, we say that class *c* has an *INV* issue if the invariant of *c* in α' does not imply the one in α. Both kinds of issues violate design-by-contract rules.

Yet another kind of generic compliance issue concerns exceptions. We say that *m* has a *THROWS* issue if for the case that the implementations α and α' agree on whether or not to throw, the thrown exceptions are different (in terms of their types or observable content). This kind of issue happens when source and target APIs use API-specific exception types or differ in the use of reusable exception types.

5.5 Domain-specific compliance issues

The generic categories are designed to fully cover all possible compliance issues. In any given API migration project, one may be able to categorize the nature of an issue at the domain level. This categorization might help in stating arguments in favor of or against

resolving certain issues, based on the given category's relevance to the application being migrated. In the sequel, we sketch two of the categories of domain-specific issues that we discovered in the study; c.f., Table 7 for illustrations.

Serialization XML can be serialized in different, semantically equivalent ways. In particular, XOM and JDOM may produce serialization results that are equivalent under XML's infoset semantics but different in terms of string-based comparison. These differences in serialization behavior are hard to neutralize by a wrapper or a transformation, but it is often easy to make applications (and their test cases) robust to such details by applying a sort of canonicalization or refraining from string-based comparison.

BaseURI XOM's 'base URI' handling is considerably more advanced than JDOM's handling. A full reproduction of XOM's semantics on top of JDOM would account for complex method implementations. However, base URI handling is rarely used in XML processing code.[5]

6 Related work

Wrapping is an established technique, in software re-engineering in particular [SM98]; legacy software is often wrapped for use in a new architecture, such as SOA [CFFT08]. We make a contribution to wrapping in so far that we leverage an API-type mapping and classification schemes for method implementations and compliance issues.

In the introduction, we already referred to related work on API migration, and our discussion was meant to reveal that all such previous work focused on *API evolution* in the sense of migrating from one version of an API to the next version. There has been effort to facilitate refactoring in API evolution [HD05,Per05,TDX07,ŞRGA08,DNMJ08]. Some of these approaches use wrapping (adapters) as an implementation technique [ŞRGA08,DNMJ08]. Those wrappers are straightforwardly derived from refactorings; in contrast, our wrappers are the actual representations of relatively heterogeneous API mappings.

Several approaches go beyond the limits of refactoring by providing some general means of transformation [CN96,KH98,BTF05,PLHM08]. Again, the showcases for all these approaches concern API evolution or migration between very much similar APIs. For instance, [BTF05] describes a rewriting-based approach for API migration that has been applied to the types `Vector` and `ArrayList` of the Java Core API, where the latter type is essentially a 'careful redesign' of the former. Nevertheless, the transformation techniques from such previous work are important ingredients of a general approach to API migration.

Our efforts to gather metadata about APIs, such as API-type mappings or compliance issues, are well in line with other recent efforts on understanding APIs at an ontology level [RJ08]. We are also inspired by other related uses of metadata in program comprehension, reverse engineering and re-engineering [BCPS05,BGGN08].

[5] Among *all* of the 43 SourceForge projects that use Subversion as repository and that use XOM, there is apparently only a single project that performs nontrivial base URI handling.

7 Conclusion

We have researched API migration with specific interest in couples of source and target APIs that were developed independently of each other. We have engineered the process of API migration in this context and reported on one study concerning two popular XML APIs of the Java platform. The various differences between the chosen APIs were identified, classified, and measured in a systematic way.

Our work shows that API migration for independently developed APIs may be manageable. Despite the many semantical and contractual differences, despite different features and designs, one can construct a reasonably compliant wrapper for API migration in a systematic, incremental, and test-driven manner. The use of a strong test suite for the API and a useful test suite for the application under migration are indeed critical. Our experiments substantiate that a wrapper-based reimplementation of an API may lack full compliance with the API's test suite, while it can be still fully compliant with the test suite of the application under migration.

One area of future work concerns the provision of a more general wrapping technique that can deal with all forms of subtyping, callbacks, and extensions points in APIs (and frameworks). We also need to generalize the described approach by applying it to other domains such as GUI or database programming.

Further, we would like to abstract from the low-level approach of specifying API migrations as metadata-annotated wrapper implementations. That is, we seek an appropriate transformation language that can perhaps even be executed in two manners: either as a source-code transformation or as a wrapper generator.

Finally, any resolved issue, say for a given method *m*, adds complexity to the API migration. A wrapper seems to hide that complexity 'inside', except perhaps for the implied performance penalty. Worse, the transformation option of API migration incurs the added complexity for every call to *m*. Hence, it is important to find an effective way of deciding on whether or not a given compliance issue needs to be dealt with for a given source location that calls *m*.

Acknowledgements This work is partially supported by IBM Centers for Advanced Studies, Toronto.

References

[Amb06] Scott W. Ambler. The Object-Relational Impedance Mismatch, 2006. `http://www.agiledata.org/essays/impedanceMismatch.html`.

[BCPS05] Marcello Bruno, Gerardo Canfora, Massimiliano Di Penta, and Rita Scognamiglio. An Approach to support Web Service Classification and Annotation. In *2005 IEEE International Conference on e-Technology, e-Commerce, and e-Services (EEE 2005), Proceedings*, pages 138–143. IEEE Computer Society, 2005.

[BDH+09] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. A foundation for flow-based program matching: using temporal logic and model checking. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 114–126. ACM, 2009.

[BGGN08] Andrea Brühlmann, Tudor Gîrba, Orla Greevy, and Oscar Nierstrasz. Enriching Reverse Engineering with Annotations. In *Model Driven Engineering Languages*

and Systems, 11th International Conference, MoDELS 2008, Proceedings, volume 5301 of *LNCS*, pages 660–674. Springer, 2008.

[BTF05] Ittai Balaban, Frank Tip, and Robert Fuhrer. Refactoring support for class library migration. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 265–279. ACM, 2005.

[CFFT08] Gerardo Canfora, Anna Rita Fasolino, Gianni Frattolillo, and Porfirio Tramontana. A wrapping approach for migrating legacy system interactive functionalities to Service Oriented Architectures. *Journal of Systems and Software*, 81(4):463–480, 2008.

[CN96] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance*, page 359. IEEE Computer Society, 1996.

[DNMJ08] Danny Dig, Stas Negara, Vibhu Mohindra, and Ralph Johnson. Reba: refactoring-aware binary adaptation of evolving libraries. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pages 441–450. ACM, 2008.

[HD05] Johannes Henkel and Amer Diwan. CatchUp!: capturing and replaying refactorings to support API evolution. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 274–283. ACM, 2005.

[KH98] Ralph Keller and Urs Hölzle. Binary component adaptation. In *ECCOP '98: Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 307–329. Springer, 1998.

[KLV05] A. S. Klusener, Ralf Lämmel, and Chris Verhoef. Architectural modifications to deployed software. *Science of Computer Programming*, 54(2-3):143–211, 2005.

[LM07] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch (Changing lead into gold). In *Spring School on Datatype-Generic Programming, Lecture Notes*, volume 4719 of *LNCS*, pages 285–367. Springer, 2007.

[Per05] Jeff H. Perkins. Automatically generating refactorings to support API evolution. In *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 111–114. ACM, 2005.

[PLHM08] Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 2008 EuroSys Conference*, pages 247–260. ACM, 2008.

[RJ08] Daniel Ratiu and Jan Juerjens. Evaluating the Reference and Representation of Domain Concepts in APIs. In *16th International Conference on Program Comprehension (ICPC 2008)*, pages 242–247. IEEE Computer Society, 2008.

[SM98] Harry M. Sneed and R. Majnar. A case study in software wrapping. In *International Conference on Software Maintenance (ICSM 1998), Proceedings*, pages 86–93. IEEE Computer Society, 1998.

[ŞRGA08] Ilie Şavga, Michael Rudolf, Sebastian Götz, and Uwe Aßmann. Practical refactoring-based framework upgrade. In *GPCE '08: Proceedings of the 7th international conference on Generative Programming and Component Engineering*, pages 171–180. ACM, 2008.

[TDX07] Kunal Taneja, Danny Dig, and Tao Xie. Automated detection of API refactorings in libraries. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated Software Engineering*, pages 377–380. ACM, 2007.

[Tho03] Dave Thomas. The Impedance Imperative: Tuples + Objects + Infosets = Too Much Stuff! *Journal of Object Technology*, 2(5):7–12, September–October 2003.