API migration is a hard problem!

Ralf Lämmel
Software Languages Team, University of Koblenz-Landau

Acknowledgement: This is joint work with (in alphabetic order) Thiago Tonelli Bartolomei (University of Waterloo, Canada), Krzysztof Czarnecki (University of Waterloo, Canada), Tijs van der Storm (CWI, Amsterdam, The Netherlands). I also acknowledge joint work within the Software Languages Team on the related subject of API (usage) analysis; special thanks are due to Ekaterina Pek.

http://www.ece.iastate.edu/seminars-and-events/api-migration-is-a-hard-problem/
http://professor-fish.blogspot.com/2012/03/should-i-declare-defeat-on-research.html
http://softlang.wikidot.com/api
API migration is a hard problem!

A Swing demo

The same demo with the SwingWT\(^1\) wrapper

Use of an improved wrapper\(^2\)

\(^1\) [http://swingwt.sourceforge.net/](http://swingwt.sourceforge.net/)

\(^2\) T.T. Bartolomei and K. Czarnecki and R. Lämmel: Compliance testing for wrapper-based API migration. Submitted.
Plan for this talk

1. **API** = Application Programming Interface
2. The very **basics** of API migration
3. The challenge of **API differences**
4. The utility of API (usage) **analysis**
5. **Studies** on API migration
6. Compliance **testing** for API migration
7. **Related work** on API migration
8. **Call to arms** on API migration
API = Application Programming Interface
### What’s an API?

<table>
<thead>
<tr>
<th>API</th>
<th>Domain</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Collections</td>
<td>Collections</td>
<td>yes</td>
</tr>
<tr>
<td>AWT</td>
<td>GUI</td>
<td>yes</td>
</tr>
<tr>
<td>Swing</td>
<td>GUI</td>
<td>yes</td>
</tr>
<tr>
<td>Reflection</td>
<td>Other</td>
<td>yes</td>
</tr>
<tr>
<td>Core XML</td>
<td>XML</td>
<td>yes</td>
</tr>
<tr>
<td>DOM</td>
<td>XML</td>
<td>yes</td>
</tr>
<tr>
<td>SAX</td>
<td>XML</td>
<td>no</td>
</tr>
<tr>
<td>log4j</td>
<td>Logging</td>
<td>no</td>
</tr>
<tr>
<td>JUnit</td>
<td>Testing</td>
<td>no</td>
</tr>
<tr>
<td>Comm.Logging</td>
<td>Logging</td>
<td>no</td>
</tr>
</tbody>
</table>

These are core and 3rd party APIs for the **Java platform**.

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
### What’s an API?

Each .NET namespace may be regarded as an API.

<table>
<thead>
<tr>
<th>Namespace</th>
<th># Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>System.Web.*</td>
<td>2327</td>
</tr>
<tr>
<td>System.Windows.*</td>
<td>●</td>
</tr>
<tr>
<td>System.ServiceModel.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Windows.Forms.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Data.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Activities.*</td>
<td>●</td>
</tr>
<tr>
<td>System.ComponentModel.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Workflow.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Xml.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Net.*</td>
<td>●</td>
</tr>
<tr>
<td>System.DirectoryServices.*</td>
<td>●</td>
</tr>
<tr>
<td>System</td>
<td>●</td>
</tr>
<tr>
<td>Microsoft.VisualBasic.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Runtime.InteropServices.*</td>
<td>●</td>
</tr>
<tr>
<td>Microsoft.JScript.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Drawing.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Runtime.Remoting.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Configuration.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Diagnostics.*</td>
<td>●</td>
</tr>
<tr>
<td>System.IO.*</td>
<td>●</td>
</tr>
<tr>
<td>System.Reflection.*</td>
<td>●</td>
</tr>
<tr>
<td>System.EnterpriseServices.*</td>
<td>●</td>
</tr>
<tr>
<td>System.CodeDom.*</td>
<td>●</td>
</tr>
</tbody>
</table>

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
API domains

• Programming domains
  ‣ XML
  ‣ GUI
  ‣ BCE (bytecode engineering)
  ‣ Testing
  ‣ ...

• Application domains
  ‣ Financial exchange
  ‣ Human resources
  ‣ ...

W.l.o.g. (?), we have been mostly addressing programming domains in our work.
The very \textit{basics} of API migration
Why to do it?

API migration = to eliminate an application's dependencies on a given API (the “original” API) and to make it depend instead on another API (the “replacement” API).

How to do it?
Incentives for API migration

• The replacement API is
  ‣ more *modern*, or
  ‣ more *typed*, or
  ‣ more *compliant*, or
  ‣ more *efficient*, or
  ‣ more *usable*, etc.

• The original API is
  ‣ *no longer supported*, or
  ‣ *too costly* in terms of license costs.

• The application must use *less APIs* per domain.

• Help with *language migration*. 
Export person list as XML document

```java
import org.jdom.*;

public static Document makeDocument(List<Person> contacts) {
    Document doc = new Document();
    Element root = new Element("contacts");
    document.addContent(root);
    for (Person p: contacts) {
        Element px = new Element("person");
        Element namex = new Element("name");
        namex = namex.setText(p.getName());
        px = px.addContent(namex);
        root = root.addContent(px);
    }
    return doc;
}
```
Export person list as XML document

```java
import org.w3c.dom.*;
import ...;

public static Document makeDocument(List<Person> contacts) {
    DocumentBuilder b = ...
    Document doc = b....;
    Element root = doc.getDocumentElement();
    for (Person p: contacts) {
        Element px = doc.createElement("person");
        Element namex = doc.createElement("name");
        namex.setTextContent(p.getName());
        px.appendChild(namex);
        root.appendChild(px);
    }
    return document;
}
```
API migration by rewriting

Original API

Replacement API

Transformation

Application

API X

Application

API Y

uses

uses
API migration by wrapping

Application uses API X

Wrapper

API X interface

Application uses API Y

Original API

Replacement API
An application in need of API migration from Vector to ArrayList

```java
public static void main(String[] args) {
    for (int count = 0; count < 5; count++)
        h1.put(new Integer(count),
            "Item " + Integer.toString(count));
    Vector v1 = new Vector();
    retrieveData(v1);
    v2 = runWizard(v1);
    Object[] dbData = new Object[v2.size()];
    v2.copyInto(dbData);
    writeToDatabase(dbData);
}
```

API migration by rewriting
Example: Vector to ArrayList

Type mapping

- Vector \rightarrow \text{ArrayList}
- Enumeration \rightarrow \text{Iterator}

Rewriting rules

\begin{tabular}{l}
\textbf{new} \ Vector(), \ unsynchronized \rightarrow \textbf{new} \ ArrayList() \\
\textbf{new} \ Vector(), \ synchronized \rightarrow \text{Collections.synchronizedList}\textbf{(new} \ \text{ArrayList()}) \\
\text{int} \ \text{Vector:receiver.size()} \rightarrow \text{int} \ \text{receiver.size()} \\
\text{Object} \ \text{Vector:receiver.firstElement()} \rightarrow \text{Object} \ \text{receiver.get(0)} \\
\text{Object} \ \text{Vector:receiver.setElementAt(} \text{Object: value, int: index}) \rightarrow \text{Object} \ \text{receiver.set(index, value)} \\
\text{void} \ \text{Vector:receiver.copyInto(} \text{Object: array}) \rightarrow \text{void} \ \text{Util.copyInto(receiver, array)} \\
\text{Enumeration} \ \text{Vector:receiver.elements()} \rightarrow \text{Iterator} \ \text{receiver.iterator()}
\end{tabular}
API migration by **wrapping**

**Example: Vector to ArrayList**

```java
public class Vector extends AbstractList implements ... {
    ArrayList adaptee;
    public Vector() {
        adaptee = new ArrayList();
    }
    public void add(Object o) {
        adaptee.add(o);
    }
    public void setSize(int ns) {
        while (adaptee.size() < ns) adaptee.add(null);
        while (adaptee.size() > ns) adaptee.remove(ns);
    }
    public Enumeration elements() {
        return new EnumerationImpl(adaptee.iterator());
    }
    ...
}

public class EnumerationImpl implements Enumeration {
    Iterator adaptee;
    EnumerationImpl(Iterator i) { this.adaptee = i; }
    public Object nextElement() { return adaptee.next(); }
    ...
}
```

API migration by wrapping
Example: XOM’s to JDOM’s Element

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

<table>
<thead>
<tr>
<th>Pros</th>
</tr>
</thead>
<tbody>
<tr>
<td>✦ Client code does not need any transformation.</td>
</tr>
<tr>
<td>✦ Existing test harness directly applies.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons/Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>✦ The original API’s interface continues to be used.</td>
</tr>
<tr>
<td>✦ Runtime overhead because of wrapping.</td>
</tr>
<tr>
<td>✦ Defensive implementation for compliance.</td>
</tr>
</tbody>
</table>

T.T. Bartolomei, K. Czarnecki, R. Lämmel, and Tijs van der Storm: Study of an API migration for two XML APIs. SLE 2009.

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
The challenge of API differences
Different XML APIs

• Some in-memory APIs
  ▸ www.w3.org/DOM
  ▸ www.jdom.org
  ▸ www.dom4j.org
  ▸ xom.nu
Different **kinds** of XML APIs

- In-memory XML APIs
  - DOM & Co.
  - JAXB & Co.
  - JAXB & Co.

- Parsing-based XML APIs
  - SAX & Co.

- Schema-Based Object Models
API differences

• Different features (capabilities)
• Different contracts (pre/postconditions)
• Different protocols (order/data-flow constraints)
• Different type mappings across type hierarchies
• ...

Any domain
Type mapping Swing : SWT

Swing-SWT

Container  →  Control

Swing

Composite

Window  →  Scrollable

JComponent

Frame

JFrame

JList

AbstractButton

JToggleButton

JCheckBox

JButton

JViewPort

JScrollPane

Widget

List

Shell

Canvas

Decorations

ActionEvent

SelectionAdapter

SelectionEvent

SelectionListener

Button

Frame.pack()

Window.pack()
Differences regarding XML APIs

• Think of *constructing*, *querying*, *updating* XML.
• Let’s see how this is done with different APIs.
• How hard would it be to do API migration?

```xml
<order>
  <product>4711</product>
  <customer>1234</customer>
  ...
</order>
```
Variations on features: *listeners*

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

JDOM (and others) do not provide an event system.
Variations on construction: void methods versus method chaining

// XOM -- construction by void methods
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);

// JDOM -- construction by method chaining
Element order =
    new Element("order").
    addContent(new Element("product").
                addContent("4711")).
    addContent(new Element("customer").
                addContent("1234"));
More variations on construction: constructors versus factory methods

// XOM -- construction with constructors
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);

// DOM -- 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);
Variations on **update**:  

**id-based** versus **index-based** update

// XOM -- **id-based** update  
order.replaceChild(oldProduct, newProduct);

// JDOM -- **index-based** update  
int index = order.indexOf(oldProduct);  
order.setContent(index, newProduct);
Variations on **preconditions**: 
**liberal** versus **strict** checking

```java
// JDOM -- *liberal* checking of preconditions
order.removeContent(product); // properly removes.
order.removeContent(product); // quietly completes.
```

```java
// XOM -- *strict* checking of preconditions
order.removeChild(product); // properly removes.
order.removeChild(product); // throws!
```
Variations on query liveness: comatose versus live query results

// XOM -- legal detachment due to comatose query results
Elements es = order.getChildElements();
for (int i=0; i<es.size(); i++)
    es.get(i).detach();

// XOM -- illegal detachment due to life query results
for (Object k : order.getChildren())
    ((Element)k).detach();

Detach all children of the order element.
Variations on *query liveness*: make live query results comatose

// JDOM -- detachment loop with up-front snapshot
Object[] es = order.getChildren().toArray();
for (Object k : es)
    ((Element)k).detach();
Variation on semantics: **detachment** versus **cloning** of structure

// JDOM -- detach and reparent
product.detach(); // detach product from order1
order2.addContent(product); // order2 is new parent

// DOM -- clone and reparent
order2.addContent(doc2.importNode(product, true)); // true is for deep cloning
The utility of API (usage) analysis
A *trivial* application using the XML API **XOM**
Complexity of the application

- 1 package
- 4 classes

Size of inner circle = LOC
Color = # Methods
(Fields are ignored.)

Thanks are due to Victor Winter / SHIFT Lab @ UNO.
Class-to-class dependencies

Thanks are due to Victor Winter / SHIFT Lab @ UNO.

White = application classes
Gray = API types
APIs used by the application

$ more .classpath

```xml
<?xml version="1.0" encoding="UTF-8"?>
<classpath>
    ...
    <classpathentry kind="lib" path="xom/xom-1.2.6.jar"/>
</classpath>
```
A *non*-trivial application using the XML API **XOM**
Complexity of the application

Top-level circles = packages
Inner circles = classes
Size of inner circle = LOC
Color = # Methods
(Fields are ignored.)

Thanks are due to Victor Winter / SHIFT Lab @ UNO.
API footprint in the application

Top-level circles = packages
Inner circles = classes
Size of inner circle = LOC
Color = # API-type references
(Fields are ignored.)

Thanks are due to Victor Winter / SHIFT Lab @ UNO.
One application class with API usage

Orange = application entities
Gray = API entities
API references are relatively localized in one method.

Thanks are due to Victor Winter / SHIFT Lab @ UNO.
API coverage by the application

Top-level circles = packages
Inner circles = classes
Size of inner circle = LOC
Color = # API-type references

Thanks are due to Victor Winter / SHIFT Lab @ UNO.

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
APIs used by the application

$ pwd
/Users/laemmel/101companies/shiftlabAnalyses/input/src/cdk/jar

$ ls
JRI.jar commons-cli-1.0.jar jgrapht-0.6.0.jar xalan.jar xpp3-1.1.4c.jar
LICENSE dtd-xercesImpl.jar jniinchi-0.5.jar xercesImpl-2.9.0.jar
antlr.jar jama-1.0.2.jar log4j.jar xml-apis.jar
antlr.jar jaxen.jar sjava-0.68.jar xom-1.1.jar
antlr.jar javacc.jar vecmath1.2-1.14.jar xom-1.2.jar
cmlxom-2.5-b1.jar jaxen.jar vecmath1.2-1.14.jar xom-1.2.jar
ctime-2.5-b1.jar-orig jaxen.jar vecmath1.2-1.14.jar xom-1.2.jar
API usage analysis of a SourceForge-based Java corpus
(Oct 2008)

• 6K+ projects from SourceForge
• 1.5K projects built (200k classes)
• 60 reference projects
• Pool of 77 known APIs aggregated
• Additional API packages detected automatically
Many APIs per project

The image shows a scatter plot with the number of used known APIs on the y-axis and project size (in NCLOC) on the x-axis. The plot includes three categories: Unbuilt projects, Built projects, and Reference projects. Each category is represented by different symbols.

The plot illustrates a trend of increasing API footprint with project size. The numbers of used known APIs and project size are shown with maxima and quartiles, indicating variability within the data.

The chart highlights that many APIs are used per project, especially as project size increases. The distribution of data points suggests that the number of APIs used grows more quickly than project size, as indicated by the logarithmic scale for both axes.

The figure also suggests that there is a higher API footprint in unbuilt projects compared to built projects. This can be inferred from the higher density of data points in the unbuilt projects category.
Many distinct API methods

As it is the case with other forms of API usage analysis, API migration and software asbestos are really exercised in one program scope or only separately.

We also need to clarify how to measure usage of API methods. In a call, a method call is counted towards the statically declared receiver type in the case of an instance call or as the hosting scope in the case of a static call. Another option is counting additionally local variable declarations or argument positions as the hosting scope in the case of a static call or as the constructor call in the case of an instance call. Yet another option is using additionally local variable declarations or argument positions as the hosting scope in the case of a static call or as the constructor call in the case of an instance call. Another option is counting imports very easily even if we cannot measure such imports very easily.

Figure 9 shows the numbers of distinct API methods used in projects without distinguishing APIs.

There is a trend of increasing API footprint with project size. Further, the number of APIs used in a project as a proxy for the difficulty of project dependence of a project. In [7], we mention such API dependence as a form of platform dependence of a project.

In fact, such footprint is really helpful in improving API usability [792, 78]. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

Figure 2. Numbers of known APIs used in the projects—known to be difficult API combinations are counted.

The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

An important form of API usage analysis concerns API coverage by the corpus. There is a trend of increasing API footprint with project size. Further, the number of APIs used in a project as a proxy for the difficulty of project dependence of a project. In [7], we mention such API dependence as a form of platform dependence of a project.

In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

Figure 9 shows the numbers of distinct API methods used in projects. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

An important form of API usage analysis concerns API coverage by the corpus. There is a trend of increasing API footprint with project size. Further, the number of APIs used in a project as a proxy for the difficulty of project dependence of a project. In [7], we mention such API dependence as a form of platform dependence of a project.

In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

Figure 9 shows the numbers of distinct API methods used in projects. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

An important form of API usage analysis concerns API coverage by the corpus. There is a trend of increasing API footprint with project size. Further, the number of APIs used in a project as a proxy for the difficulty of project dependence of a project. In [7], we mention such API dependence as a form of platform dependence of a project.

In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

Figure 9 shows the numbers of distinct API methods used in projects. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

An important form of API usage analysis concerns API coverage by the corpus. There is a trend of increasing API footprint with project size. Further, the number of APIs used in a project as a proxy for the difficulty of project dependence of a project. In [7], we mention such API dependence as a form of platform dependence of a project.

In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

Figure 9 shows the numbers of distinct API methods used in projects. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

An important form of API usage analysis concerns API coverage by the corpus. There is a trend of increasing API footprint with project size. Further, the number of APIs used in a project as a proxy for the difficulty of project dependence of a project. In [7], we mention such API dependence as a form of platform dependence of a project.

In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.

Figure 9 shows the numbers of distinct API methods used in projects. The APIs or API methods used in a project provide insight into the API-related complexity of the project. In fact, such footprint is really helpful in improving API usability. That is, coverage information is helpful in improving API usability. That is, coverage may be considered for either a specific project or a particular API.
Low usage of API methods

Cumulative coverage of API methods is 24.10% (if we ignore JDOM sources).
Few interfaces implementations for APIs

java.util.Iterator

java.awt.event.ActionListener

Only 7 APIs are used in more than 10 projects with “generalization”.

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
Few subclass derivations for APIs

Figure 3: Tag cloud of overridden API classes.

Only 7 APIs are used in more than 10 projects with "generalization".

Figure 4: Frequency of calling API vs non-API methods.
API usage analysis of an open-source .NET corpus (June 2011)

- 17 projects from different repositories
- well known, widely used, mature
- amenable to dynamic analysis
- API = .NET namespace
- 401 in total
- 69 after grouping
Corpus of the .NET study

<table>
<thead>
<tr>
<th>.NET</th>
<th>Project</th>
<th>Repository</th>
<th>LOC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.5</td>
<td>Castle ActiveRecord</td>
<td>GitHub</td>
<td>30,303</td>
<td>Object-relational mapper</td>
</tr>
<tr>
<td>4.0</td>
<td>Castle Core Library</td>
<td>GitHub</td>
<td>36,659</td>
<td>Core library for the Castle framework</td>
</tr>
<tr>
<td>3.5</td>
<td>Castle MonoRail</td>
<td>GitHub</td>
<td>58,121</td>
<td>MVC Web framework</td>
</tr>
<tr>
<td>4.0</td>
<td>Castle Windsor</td>
<td>GitHub</td>
<td>50,032</td>
<td>Inversion of control container</td>
</tr>
<tr>
<td>4.0</td>
<td>Json.NET</td>
<td>Codeplex</td>
<td>43,127</td>
<td>JSON framework</td>
</tr>
<tr>
<td>2.0</td>
<td>log4net</td>
<td>Sourceforge</td>
<td>27,799</td>
<td>Logging framework</td>
</tr>
<tr>
<td>2.0</td>
<td>Lucene.Net</td>
<td>Apache.org</td>
<td>158,519</td>
<td>Search engine</td>
</tr>
<tr>
<td>4.0</td>
<td>Managed Extensibility Framework</td>
<td>Codeplex</td>
<td>149,303</td>
<td>Framework for extensible applications and components</td>
</tr>
<tr>
<td>4.0</td>
<td>Moq</td>
<td>GoogleCode</td>
<td>17,430</td>
<td>Mocking library</td>
</tr>
<tr>
<td>2.0</td>
<td>NAnt</td>
<td>Sourceforge</td>
<td>56,529</td>
<td>Build tool</td>
</tr>
<tr>
<td>3.5</td>
<td>NHibernate</td>
<td>Sourceforge</td>
<td>330,374</td>
<td>Object-relational mapper</td>
</tr>
<tr>
<td>3.5</td>
<td>NUnit</td>
<td>Launchpad</td>
<td>85,439</td>
<td>Unit testing framework</td>
</tr>
<tr>
<td>4.0</td>
<td>Patterns &amp; Practices - Prism</td>
<td>Codeplex</td>
<td>146,778</td>
<td>Library to build flexible WPF and Silverlight apps</td>
</tr>
<tr>
<td>3.5</td>
<td>RhinoMocks</td>
<td>GitHub</td>
<td>23,459</td>
<td>Mocking framework</td>
</tr>
<tr>
<td>2.0</td>
<td>SharpZipLib</td>
<td>Sourceforge</td>
<td>25,691</td>
<td>Compression library</td>
</tr>
<tr>
<td>2.0</td>
<td>Spring.NET</td>
<td>GitHub</td>
<td>183,772</td>
<td>Framework for enterprise applications</td>
</tr>
<tr>
<td>2.0</td>
<td>xUnit.net</td>
<td>Codeplex</td>
<td>23,366</td>
<td>Unit testing framework</td>
</tr>
<tr>
<td>Namespaces</td>
<td>Projects</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rows: top-10 .NET namespaces, in terms of number of types.
Middle block of columns: actual reuse by project of the corpus.
Leftmost column: potential reuse in terms of specializability.
Rightmost column: summary of actual reuse.
For each class, the number of subclasses in the corpus are shown.

Figure 9. Top 30 .NET classes inherited in the corpus

For each interfaces, the number of implementing classes in the corpus are shown. Note: the full bar counts all implementations whereas the black part excludes classes that can be reliably classified as being compiler-generated.

Figure 10. Top 30 .NET interfaces implemented in the corpus
For each class, the number of subclasses in the corpus are shown.

Figure 9. Top 30 .NET classes inherited in the corpus

For each interfaces, the number of implementing classes in the corpus are shown. Note: the full bar counts all implementations whereas the black part excludes classes that can be reliably classified as being compiler-generated.

Figure 10. Top 30 .NET interfaces implemented in the corpus

Top-30 implemented .NET interfaces
Studies on API migration


• **Wrapper-based studies**
  - **XML**: Re-implement XOM in terms of JDOM
  - **GUI**: Re-implement Swing in terms of SWT and v.v.
  - **GUI**: Improve the existing SwingWT wrapper
  - **BCE**: Re-implement BCEL in terms of ASM

• **Wrapper engineering**
  - **Validation** based on suitable applications
  - **Tracking** of engineering process and metrics
  - Use of **design patterns** for wrapper design
  - Use of **compliance testing**

See the papers for details!
Wrapper-based reimplementation of XOM in terms of JDOM
XOM versus JDOM metrics
### Types: XOM versus JDOM

<table>
<thead>
<tr>
<th>API package</th>
<th>#Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>nu.xom</td>
<td>35</td>
</tr>
<tr>
<td>nu.xom.canonical</td>
<td>2</td>
</tr>
<tr>
<td>nu.xom.converters</td>
<td>2</td>
</tr>
<tr>
<td>nu.xom.xinclude</td>
<td>9</td>
</tr>
<tr>
<td>nu.xom.xslt</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>50</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>API package</th>
<th>#Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.jdom</td>
<td>22</td>
</tr>
<tr>
<td>org.jdom.adapters</td>
<td>8</td>
</tr>
<tr>
<td>org.jdom.filter</td>
<td>4</td>
</tr>
<tr>
<td>org.jdom.input</td>
<td>5</td>
</tr>
<tr>
<td>org.jdom.output</td>
<td>7</td>
</tr>
<tr>
<td>org.jdom.transform</td>
<td>4</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>50</strong></td>
</tr>
</tbody>
</table>

The wrapper is mainly concerned with the “core” packages.

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
Features = instance methods, static methods, constructors
CDK -- the application for validation

The Chemistry Development Kit (CDK) is a Java library for structural chemo- and bioinformatics. It is now developed by more than 50 developers all over the world and used in more than 10 different academic as well as industrial projects world wide.

About the Chemistry Development Kit (CDK)

The CDK design originated from a meeting on September 27-29 2000, Christoph Steinbeck, Egon Willighagen and Dan Gezelter met at Notre Dame University, South Bend, USA, to discuss the architecture of the package. The first actual implementation very much relied on the on the earlier work by the lab of Christoph Steinbeck and his older CompChem libraries, which urgently needed a rewrite. On the
Establish the API correspondence

<table>
<thead>
<tr>
<th>nu.xom</th>
<th>org.jdom</th>
<th>regular features</th>
<th>irregular features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Attribute</td>
<td>25</td>
<td>3</td>
</tr>
<tr>
<td>Builder</td>
<td>input.SAXBuilder</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>Comment</td>
<td>Comment</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>Document</td>
<td>Document</td>
<td>26</td>
<td>2</td>
</tr>
<tr>
<td>Element</td>
<td>Element</td>
<td>51</td>
<td>2</td>
</tr>
<tr>
<td>Elements</td>
<td>java.util.List</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Node</td>
<td></td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Nodes</td>
<td>java.util.List</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>ProcessingInstruction</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td>Text</td>
<td>Text; CDATA</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>168</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 5.

Table 6.

Table 7.

Table 8.

The general goal of —PI migration is to eliminate usage of a given —PI use the excellent XOM test suite to this compliance of the —PI implemetation of XOM with the ac.

4. Wrapper-based API migration

4.2. Wrapper implementation

We prepare an "empty" —PI implementation of the source —PI instead.

There is one abstract but still regular source type: #OM case study. -ll but one source type

Node

Elements

Nodes

ProcessingInstruction

Text

Attribute

Builder

Comment

Document

Element

Elements

Node

Nodes

ProcessingInstruction

Text

org.jdom

#regular features

#irregular features

25

3

15

1

12

2

26

2

51

2

2

2

8

1

14

4

13

1

168

26

25

3

15

1

12

2

26

2

51

2

2

2

8

1

14

4

13

1

168

26

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
Correspondence graph for core APIs: XOM vs. JDOM
Correspondence graph for core APIs: XOM vs. JDOM
Correspondence graph for core APIs: XOM vs. JDOM
Correspondence graph for the two Comment types
Correspondence graph for the two \textit{Attribute} types
Correspondence graph for the two **Attribute** types
Correspondence graph for the two \textbf{Attribute} types
4 levels of adaptation

1. Straightforward delegation
2. Argument / result processing, conditional delegation
3. Implementation of a missing feature (be it as a “macro”)
4. Re-implementation of a feature due to semantical mismatch
Adaptations per level

<table>
<thead>
<tr>
<th>nu.xom</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>16</td>
<td>7</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Builder</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Comment</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Document</td>
<td>15</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Element</td>
<td>22</td>
<td>16</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>Elements</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Node</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Nodes</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ProcessingInstruction</td>
<td>15</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Text</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>87</td>
<td>51</td>
<td>21</td>
<td>11</td>
</tr>
</tbody>
</table>
Compliance statistics
Compliant and non-compliant test cases

<table>
<thead>
<tr>
<th>Test suite</th>
<th>#Test cases</th>
<th>#Compliant test cases</th>
<th>#Non-compliant test cases</th>
<th>API Method coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>697</td>
<td>417</td>
<td>280</td>
<td>156</td>
</tr>
<tr>
<td>App</td>
<td>752</td>
<td>752</td>
<td>0</td>
<td>35</td>
</tr>
</tbody>
</table>
## Methods
in compliance with different test suites at different levels

<table>
<thead>
<tr>
<th>Test suite</th>
<th>always</th>
<th>sometimes</th>
<th>never</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>API</strong></td>
<td>75</td>
<td>77</td>
<td>4</td>
<td>79</td>
</tr>
<tr>
<td><strong>Application</strong></td>
<td>35</td>
<td>0</td>
<td>0</td>
<td>200</td>
</tr>
</tbody>
</table>
Generic compliance issues

- **PRE**: wrapper precondition too strong (violates DBC)
- **PRE**: wrapper precondition too weak (too little checking)
- **POST**: wrapper postcondition too weak (violates DBC)
- **INVARIANT**: wrapper invariant deviates
- **THROWS**: exception types or values deviate

<table>
<thead>
<tr>
<th>Type</th>
<th>Feature</th>
<th>Issue type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>toXML()</td>
<td>Post</td>
<td>JDOM’s escaping is different from XOM’s</td>
</tr>
<tr>
<td></td>
<td>Attribute(String,String)</td>
<td>Pre</td>
<td>XOM allows colonized names in the first argument whereas JDOM does not</td>
</tr>
<tr>
<td>DocType</td>
<td>setSystemID(String)</td>
<td>Invariant</td>
<td>XOM requires that Public ID has been nulled before SystemID can be</td>
</tr>
<tr>
<td>Element</td>
<td>addAttribute(Attribute)</td>
<td>Throws</td>
<td>XOM throws MultipleParentException if argument is parented whereas JDOM throws IllegalAddException</td>
</tr>
</tbody>
</table>
API-specific compliance issues

- Escaping of characters serialization
- Formatting details of serialization
- Handling of namespace declarations
- Handling of DocType/embedded DTD
- ...
Compliance testing for API migration

API migration is a hard problem!

- How to spot such differences?
- What contracts to use this end?
- How to avoid regressions?
A scenario is an execution of the application using the original API. The interpreter of the method enriches scenarios—both with assertions related to application logic, which may not also check (indirectly) some aspects of API usage. Tests for an application, for selected API contracts.

Interpretation of the trace is checked to reproduce the captured behavior. We also include the event types in the metadata associated with the tracer report. In particular, each incoming event is associated with an event type.

There are additional type rules needed for rewriting type names (e.g., regular expressions over their fully qualified names). The result is a validated trace that can be used as input for the wrapper development phase.

**Trace Collection**
- Scenario Design
- Automated Scenario
- Original API
- Type Rules
- Tracer Configuration
- Trace Complete?

**Trace Validation**
- Violation?
- Yes: Interpreter Configuration
- No: Interpreter Original API Mode
- Validated Trace
- Type Rules
- Settings

**Wrapper Development**
- Good Enough?
- Yes
- No: Evolution Needed?
- No
- Wrapper Evolution
- Wrapper
- Replacement API
- Configuration Refinement
- Automated Task
- Manual Task
- Refinable Artifact
- Artifact
A scenario is an execution of the application using the original API. We also refer to these contracts as 'basic contracts'. With the interpreter of the method, scenarios—both with and without assertions for application logic—with assertions related to API usage may not also check (indirectly) some aspects of API usage.

Scenarios usually target the applications at hand (as opposed to the code engineering framework). For instance, the interaction between application and API may be mediated by auxiliary types by default, thereby being able to mock multiple parts of the application.

Tracing
- The tracer executes the scenarios and collects traces from the application.
- Traces that are derived from scenarios must not depend on unbound object references in API calls, and developers must also be incorporated into the traces. The tracer reports on unbound object references in API calls, and developers should not be migrated.
- Traces must be validated before they are even attempted. In particular, each incoming event is associated with the type of the object it is attached to.
- Additional type rules are needed for rewriting type names such as java.awt.*, javax.swing.*, javax.accessibility.*, java.lang.Runtime, and javax.swing.event.*. Additional rules were needed: one scenario needed 8 auxiliary types but most did not need any.

Interpretation
- The interpretation of the trace is checked to reproduce the capabilities of the original API. However, the interaction may need to be enriched with metadata that belong to the application and the API; this metadata is also incorporated into the traces.

Type Rules
- Type rules were needed: one scenario needed 8 auxiliary types but most did not need any. Rules were needed: one scenario needed 8 auxiliary types but most did not need any.

5.1 Phase: Trace Collection
- The tracer executes the scenarios and collects traces from the application.
- Traces that are derived from scenarios must not depend on unbound object references in API calls, and developers must also be incorporated into the traces. The tracer reports on unbound object references in API calls, and developers should not be migrated.
- Traces must be validated before they are even attempted. In particular, each incoming event is associated with the type of the object it is attached to.
- Additional type rules are needed for rewriting type names such as java.awt.*, javax.swing.*, javax.accessibility.*, java.lang.Runtime, and javax.swing.event.*. Additional rules were needed: one scenario needed 8 auxiliary types but most did not need any.

5.2 Phase: Trace Validation
- The interpretation of the trace is checked to reproduce the capabilities of the original API. However, the interaction may need to be enriched with metadata that belong to the application and the API; this metadata is also incorporated into the traces. The tracer reports on unbound object references in API calls, and developers should not be migrated.
- Traces must be validated before they are even attempted. In particular, each incoming event is associated with the type of the object it is attached to.
- Additional type rules are needed for rewriting type names such as java.awt.*, javax.swing.*, javax.accessibility.*, java.lang.Runtime, and javax.swing.event.*. Additional rules were needed: one scenario needed 8 auxiliary types but most did not need any.
A trace for a Swing scenario

\[ e_1, t_0 \rightarrow \_\_Robot.<init>():0 \ldots \]
\[ e_{16}, t_0 \rightarrow \_\_ToolTipDemo.<init>(\_):1 \ldots \]
\[ e_{22}, t_0 \rightarrow \_\_ToolTipDemo$Cow.<init>(1):3 \ldots \]
\[ e_{29}, t_0 \rightarrow \_\_Polygon.<init>():4 \ldots \]
\[ e_{32}, t_0 \rightarrow 3:ToolTipDemo$Cow.setToolTipText("Cow"):V \ldots \]
\[ e_{62}, t_0 \rightarrow 2:JPanel.add(3):3 \ldots \]
\[ e_{66}, t_0 \rightarrow \_\_Frame.<init>("ToolTip Demo"):5 \ldots \]
\[ e_{68}, t_0 \rightarrow 5:Frame.getContentPane():6 \]
\[ e_{72}, t_0 \rightarrow 6:Container.add(2, "Center"):\_\_ \ldots \]
\[ e_{73}, t_0 \rightarrow 5:Frame.show():V \ldots \]
\[ e_{85}, t_0 \rightarrow 0:Robot.mouseMove(‘342’, ‘312’):V \]
\[ e_{86}, t_0 \rightarrow \_\_Thread.sleep(‘3000’):V \]
\[ e_{87}, t_1 \leftarrow 6:Container.contains(‘155’, ‘112’):’true’ \]
\[ e_{88}, t_1 \rightarrow \_\_Point.<init>(‘155’, ‘112’):7 \]
\[ e_{89}, t_1 \rightarrow 4:Polygon.contains(7):’true’ \]
\[ e_{90}, t_1 \rightarrow 3:ToolTipDemo$Cow.setToolTipText("<html><center><font color=blue size=+2>Mooooooo</font></center></html>"):V \]
A scenario is an execution of the application using the original API. We also refer to these contracts as 'basic contracts'. With these scenarios, we can verify the compliance of new APIs with the original contracts.

Automated testing can be performed by using a record&replay tool (such as GUITAR). However, challenges still arise. Consider, for example, API migration in the context of API migration, this requirement may imply extra efforts. For selected API contracts, this is a common requirement for automated testing. In the case of API migration, this is particularly important.

For GUI APIs: GUI actions would need to be automated for external actions, and all results should be reproducible. It is the responsibility of the developer to design scenarios that cover the desired uses of the application since they represent the requirements for the migration. The wrapper will be considered compliant if it properly replaces the original API in the context of the application.

5.1 Phase: Trace Collection

- The tracer executes the scenarios and collects traces.
- Traces derived from scenarios must not depend on external actions, and all results should be reproducible. This also helps with automating the scenario and making it easier to validate.

5.2 Phase: Trace Validation

- Traces must be validated before they are even attempted. This involves checking the interpretation of the trace to reproduce the captured behavior. We also include the events for such auxiliary types in the selected contracts (see 'Settings' in Fig. 6) can be incorporated into the traces.

The interpreter builds a model of all types and methods from the trace so that mock application types and methods can be provided for trace execution; also, assertions for expected methods can be provided. In particular, each incoming event is associated with a mock application type; each mock application method can be executed. In particular, state for checking state integrity; see the corresponding aggregation of state that is captured in the trace.

5.3 Phase: Wrapper Development

- The interpreter builds a model of all types and methods from the trace so that mock application types and methods can be provided for trace execution; also, assertions for expected methods can be provided. In particular, each incoming event is associated with a mock application type; each mock application method can be executed. In particular, state for checking state integrity; see the corresponding aggregation of state that is captured in the trace.

The interpreter builds a model of all types and methods from the trace so that mock application types and methods can be provided for trace execution; also, assertions for expected methods can be provided. In particular, each incoming event is associated with a mock application type; each mock application method can be executed. In particular, state for checking state integrity; see the corresponding aggregation of state that is captured in the trace.
Contracts

- Return values
- Return ids (modulo isomorphism)
- Callbacks
- Set/get contracts

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Observer</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>setX (\ldots v)</td>
<td>getX()</td>
<td>getter returns (v)</td>
</tr>
<tr>
<td>setX(\text{boolean} (v))</td>
<td>isX()</td>
<td>predicate returns (v)</td>
</tr>
<tr>
<td>addX (\ldots v)</td>
<td>getXs()</td>
<td>result contains (v)</td>
</tr>
<tr>
<td>add((T v))</td>
<td>getTs()</td>
<td>result contains (v)</td>
</tr>
<tr>
<td>put (\ldots k, \ldots v)</td>
<td>get(k)</td>
<td>getter returns (v)</td>
</tr>
</tbody>
</table>

- Apply “tunings”
Tuning

DSL-based declaration of the tuning
org/apache/bcel/classfile/
   JavaClass.getBytes : BytecodeTuning

Match operation for the tuning

```java
public class BytecodeTuning implements ... {
   public boolean matches(byte[] expected,
                           byte[] actual) {
       return serialize(expected).equals(
                           serialize(actual));
   }
   private static String serialize(byte[] bytecode)
   StringWriter w = new StringWriter();
   new ClassReader(bytecode).accept(
       new TraceClassVisitor(new PrintWriter(w)), 0);
   return w.toString();
}
```
Related work on API migration
Related work -- selection

• “Representative” of much work on API upgrade
• Orthogonal aspects:
  ‣ Inference of refactoring
  ‣ Wrapper generation
• General API migration ≠ refactoring
new Vector(), unsynchronized
new Vector(), synchronized
int Vector:receiver.size()
Object Vector:receiver.firstElement()
Object Vector:receiver.setElementAt(Object: value, int: index)
void Vector:receiver.copyInto(Object: array)
Enumeration Vector:receiver.elements()

→ new ArrayList()
→ Collections.synchronizedList(new ArrayList())
→ int receiver.size()
→ Object receiver.get(0)
→ Object receiver.set(index, value)
→ void Util.copyInto(receiver, array)
→ Iterator receiver.iterator()

- **1:1 type mapping**
  - Type constraints for applicability
  - Escape analysis for synchronization
  - No mapping for exception types
  - No mapping for application subtypes

- **Method mapping**
  - Unidirectional mapping
  - Single call scope for input
Type constraints for applicability

- Consider a constructor call to JComboBox:
  Vector v = ...;
  JComboBox b = new JComboBox(v);
- Here, JComboBox belongs to the Swing library.
- There, is no constructor that takes an ArrayList.
- Hence, Vector cannot be migrated to ArrayList.

In principle, one may use coercions in these contexts.
On the issue of exception types

- **nu.xom:**
  - 28 distinguished exception types
  - Uses standard exception types

- **org.jdom:**
  - 8 distinguished exception types
  - Uses standard exception types

Without exception type mapping, applications do not even compile.
On the issue of unidirectional mappings

void Vector:receiver.copyInto(Object: array) → void Util.copyInto(receiver, array)

```java
public class Util {
    public static void copyInto(ArrayList v, Object[] target) {
        if (target == null)
            throw new NullPointerException();
        for (int i = v.size() - 1; i >= 0; i--)
            target[i] = v.get(i);
    }
}
```

What would it take to execute the rule backwards?
On the issue of single call scope

```java
public static Document makeDocument(List<Person> contacts) {
    Document doc = new Document();
    Element root = new Element("contacts");
    document.addContent(root);
    for (Person p: contacts) {
        Element px = new Element("person");
        Element namex = new Element("name");
        namex = namex.setText(p.getName());
        px = px.addContent(namex);
        root = root.addContent(px);
    }
}
```

Suppose that this code needs to rewritten to factory-based style. A flow analysis must be employed to determine `doc` as the factory.
On the issue of single call scope

```java
public static Document loadDocument(String filename) {
    Document doc = null;
    try {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        doc = builder.parse(new File(filename));
    } catch (SAXException e) {
    } catch (ParserConfigurationException e) {
    } catch (IOException e) { ... }
    return doc;
}
```

The shown code is “typical” idiomatic DOM legacy, which needs to be folded into just a few method calls, when using other XML APIs.
Interpret “semantic patches” as program transformations

- C programs as processed as control-flow grammar
- Semantic patch is interpreted as CTL formula
- Code isomorphisms and “…” enabled

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
[PadioleauLHM08] Semantic patches

Context

Addition

Removal

Any fragment of CFG

© 2009-2012 Ralf Lämmel, Software Languages Team and collaborators; see 1st page of slide deck
[PadioleauLHM08] Semantic patches

Challenge: applying [PadioleauLHM08] to an OO language and to an OO API.

---

**Mapping**

<table>
<thead>
<tr>
<th>Vector ()</th>
<th>{ return new Vector(); }</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList ()</td>
<td>{ return new ArrayList(); }</td>
</tr>
<tr>
<td>Enumeration (Vector v)</td>
<td>{ return v.elements(); }</td>
</tr>
<tr>
<td>Iterator (ArrayList a)</td>
<td>{ return a.iterator(); }</td>
</tr>
<tr>
<td>boolean (Enumeration e)</td>
<td>{ return e.hasMoreElements(); }</td>
</tr>
<tr>
<td>boolean (Iterator i)</td>
<td>{ return i.hasNext(); }</td>
</tr>
<tr>
<td>Object (Enumeration e)</td>
<td>{ return e.nextElement(); }</td>
</tr>
<tr>
<td>Object (Iterator i)</td>
<td>{ return i.next(); }</td>
</tr>
</tbody>
</table>

---

**Syntax**

\[
M ::= \emptyset \quad \text{(blank mapping)} \\
| \begin{align*}
& [ T \ ( F ) \ { B } ] \quad \text{(replacement)} \\
& \quad T \ ( F ) \ { B } \\
\end{align*} \\
| M ; M \quad \text{(list of replacements)} \\
\]

\[
F ::= \emptyset \ | \ F , \ T \ x \quad \text{(formals)} \\
\]

\[B \in \text{Java method bodies} \]

\[T \in \text{Java types} \]

\[x \in \text{Java identifiers} \]
Two forms of adaptation

- Shallow adaptation: *apply mapping directly*
- Deep adaptation: *go through mapping-defined API*

Limitations

- No formal definition available
- No use of control-flow/data-flow analysis
- Validation for some subset of Twitter/Facebook APIs
Call to arms
on API migration
Call to arms

- Name- and type-based API matching
- Flow-aware patterns
- Sound use of non-invariant type mappings
- IDE support
- Incremental development
- Test-driven development
- End-to-end validation
- Generalization beyond API migration
- Uniformity across wrapping and rewriting
That’s it.
Thanks!

http://www.ece.iastate.edu/seminars-and-events/api-migration-is-a-hard-problem/
http://professor-fish.blogspot.com/2012/03/should-i-declare-defeat-on-research.html
http://softlang.wikidot.com/api