Approaching the API migration challenge

Ralf Lämmel (Uni Koblenz)

Joint work with **Tijs van der Storm** (CWI, Amsterdam)

Further acknowledgements for the mentioned API-usage analysis: Ruwen Hahn & Jürgen Starek (Uni Koblenz)

API usage as a form of software asbestos

C++ code for window creation in pre-.NET

```
HWND hwndMain = CreateWindowEx( 0,
        "MainWinClass",    "Main Window",
        WS_OVERLAPPEDWINDOW | WS_HSCROLL | WS_VSCROLL,
        CW_USEDDEFAULT, CW_USEDEFAULT,
        CW_USEDDEFAULT, CW_USEDEFAULT,
        (HWND)NULL,(HMENU)NULL, hInstance, NULL);
ShowWindow(hwndMain, SW_SHOWDEFAULT);
UpdateWindow(hwndMain);
```

.NET version

```
Form form = new Form();
form.Text = "Main Window";
form.Show();
```

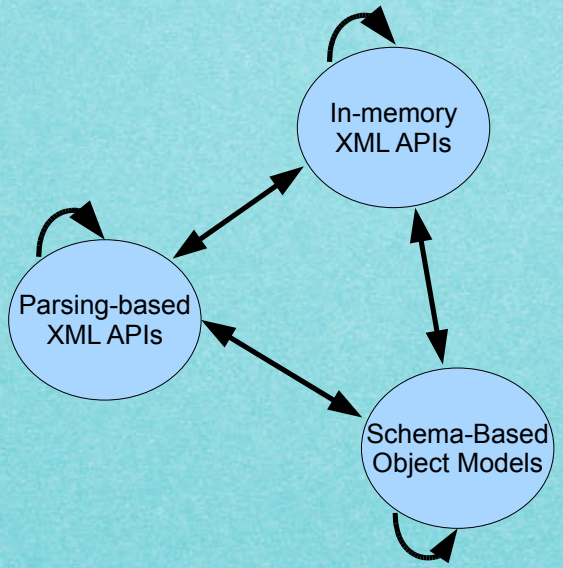
The API migration problem

Given a couple of “reasonably similar” APIs *A* and *B*, provide a transformational approach to the semi-automatic replacement of the use of *A* by the use of *B* in any given software project (say, *P*).

An illustrative domain for API migration

- Some XML APIs
  - http://www.w3.org/DOM/
  - www.jdom.org/
  - www.dom4j.org/
  - http://xom.nu/
  - JAXB (not exactly an API)

API migration for XML APIs



API migration - Why?

- Code modernization - transition to modern API
- Code hardening w.r.t. constraints offered by API
- API retirement - obsoletion of an aging API
- Complexity reduction w.r.t. number of used APIs
- API evolution - obsoletion of prior version
- Part of platform migration, MDD enabling
- ...

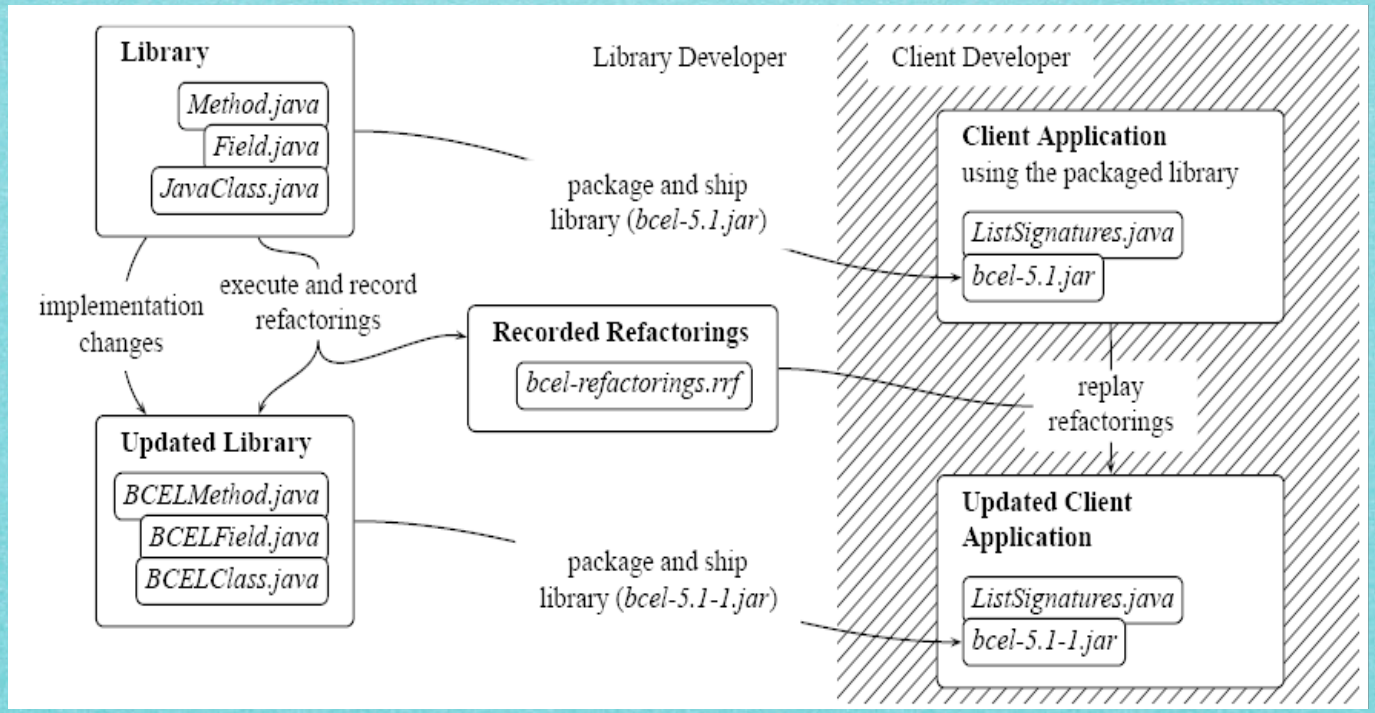
Paths for API migration

- Manual migration using guides
- Refactoring-based
- Wrapper-based re-implementation
- Wrapping + Partial Evaluation
- Special purpose transformation
  - Source code
  - Byte code

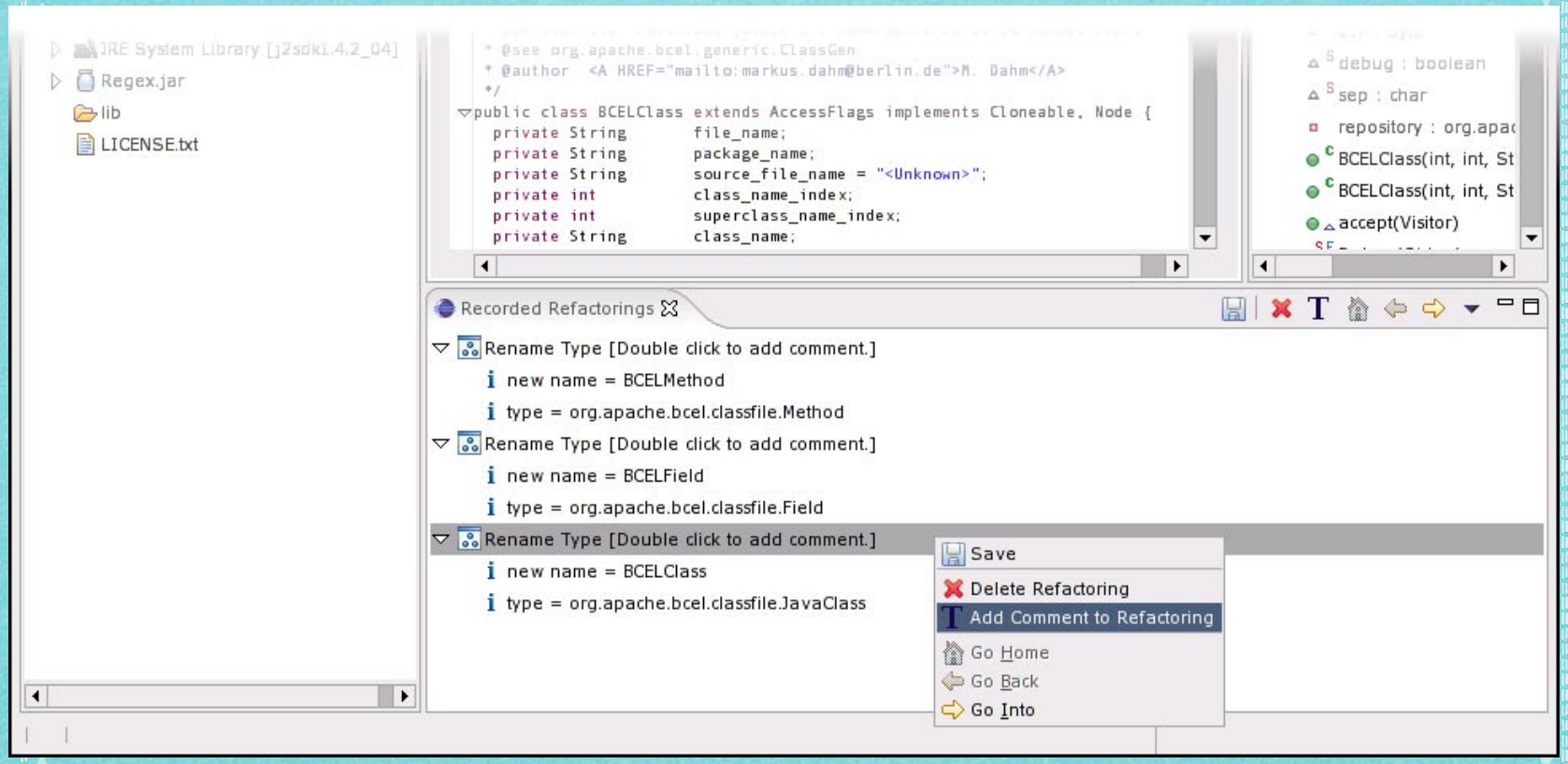
The refactoring-based path to API migration

- Pick source- or byte-code transformation.
- Derive *B* by refactoring *A* mechanically.
- Record refactorings in a script *S*.
- Deploy *B* together with *S*.
- Project *P* will be converted by replaying *S*.

Henkel, Diwan's: CatchUp! Architecture



Henkel, Diwan's: CatchUp! View for recorded refactorings



Limitations of refactoring

- Assume *A* and *B* are "independent" APIs:
  - *A*'s code must be refactored into *C* such that:
    - *C* agrees with interface of *B*.
    - *C* is observably equivalent to *B*.
  - This entails a proof of program equivalence.
- Assume *B* is successor version of *A*:
  - The available refactorings may be insufficient.
  - *B* may actually be a re-implementation of *A*.

Scenario: construct XML tree from collection of persons

```
<contacts>
   <person>
      <name>Barack Obama</name>
      <age>47</age>
   </person>
   <person>
      <name>John McCain</name>
      <age>72</age>
   </person>
</contacts>
```

The "builder scenario"

The builder scenario using the jdom API

```
public static Document makeDocument(List<Person> contacts) {
    Document document = new Document();
    Element root = new Element("contacts");
    document.addContent(root);
    for (Person p: contacts) {
        Element person = new Element("person");
        Element name = new Element("name");
        name.setText(p.getName());
        person.addContent(name);
        Element age = new Element("age");
        age.setText(new Integer(p.getAge()).toString());
        person.addContent(age);
        root.addContent(person);
    }
    return document;
} // done
```

The builder scenario using (W3C's) dom API

```
public Document makeDocument(List<Person> contacts) {
    Document document = getDomImplementation().createDocument(
                null, "contacts", null);
    Element root = document.getDocumentElement();
    for (Person p: contacts) {
        Element person = document.createElement("person");
        Element name = document.createElement("name");
        Node nameText = document.createTextNode(p.getName());
        name.appendChild(nameText);
        person.appendChild(name);
        Element age = document.createElement("age");
        Node ageText = document.createTextNode(
          new Integer(p.getAge()).toString());
        age.appendChild(ageText);
        person.appendChild(age);
        root.appendChild(person);
    }
    return document;
} // done
```

API differences

- Node construction
  - dom: uses document object as factory
  - jdom: leverages regular constructors
- Text content
  - dom: uses designated objects for text nodes
  - jdom: represents text as strings
- Construction of the document also differs ...

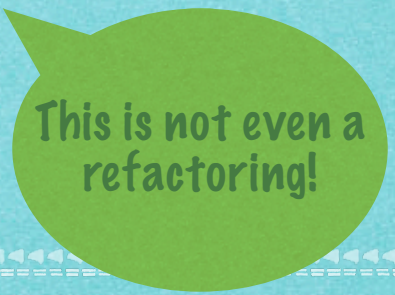
Refactoring under attack
jdom to dom

- Before: `Element person = new Element("person");`
- After: `Element person = document.createElement("person");`
- Refactoring attempt
  - Introduce instance method createElement.
    - Implement method as factory method for Element.
  - Replace constructor call by call to createElement.
    - **A suitable Document instance is needed.**
  - Hide constructor in interface.

Not a common refactoring!

Refactoring under attack
dom to jdom

- Before: `Element person = document.createElement("person");`
- After: `Element person = new Element("person");`
- Refactoring attempt
  - Replace call to createElement by constructor call.
    - **Elements no longer owned by a document.**
  - Eliminate createElement method.

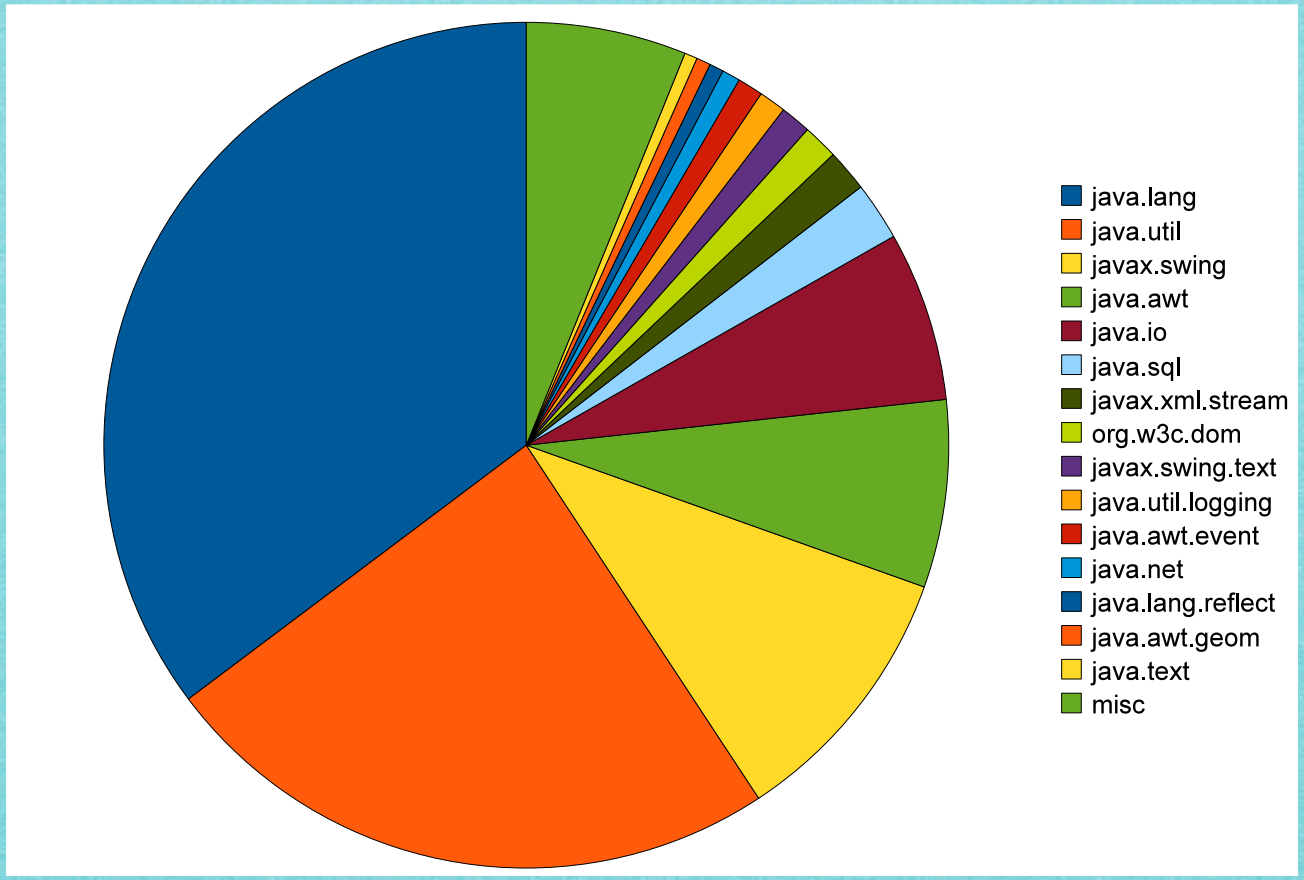


Intermezzo: Inform migration research by API usage analysis

- What is the scale of usage frequency for APIs?
- How are APIs used in a typical project?
- What combinations of APIs occur together?
- How frequent are principled usage scenarios?
- How frequent are problematic uses?
- ...

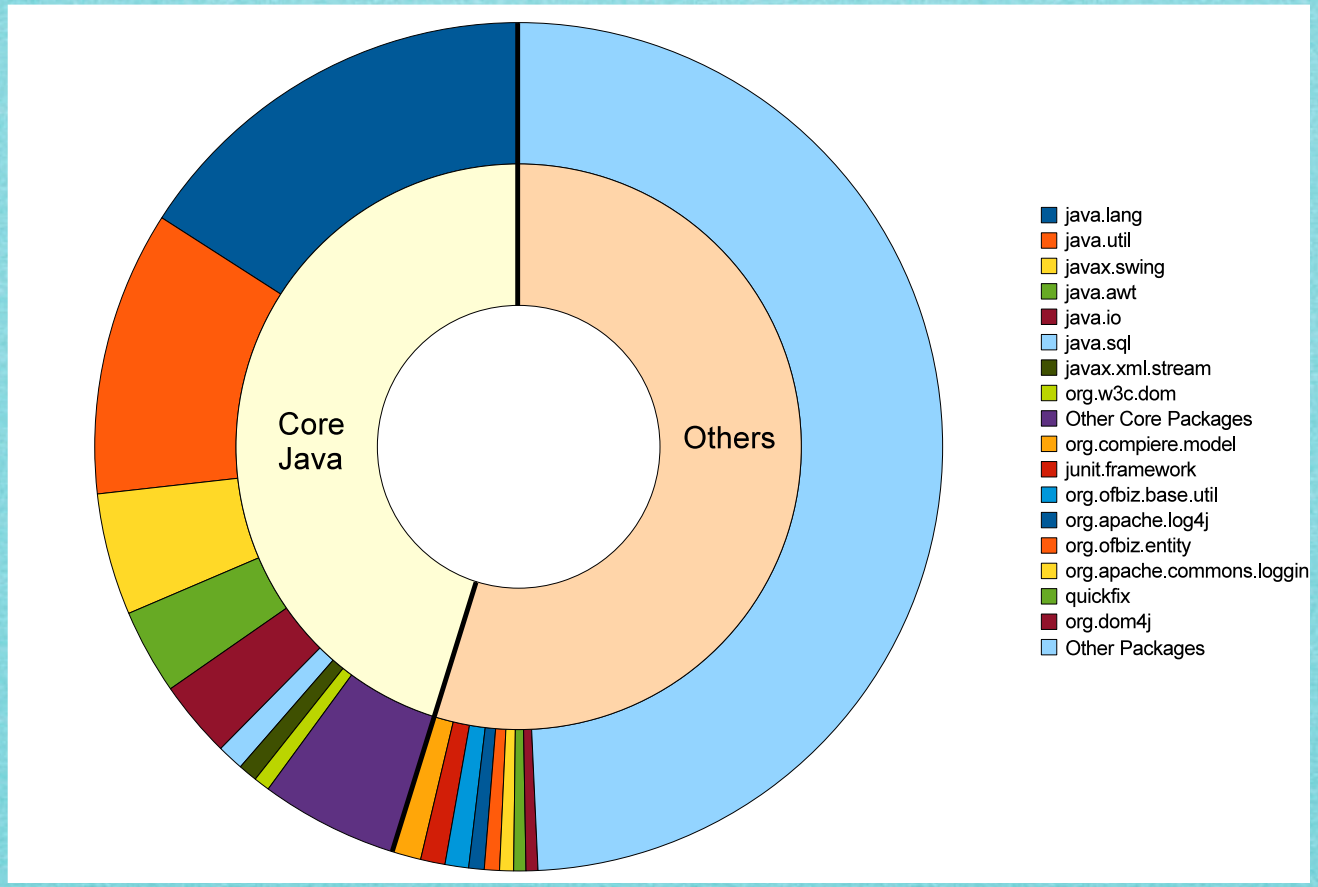
Most used Core Java packages

(based on our SourceForge study)



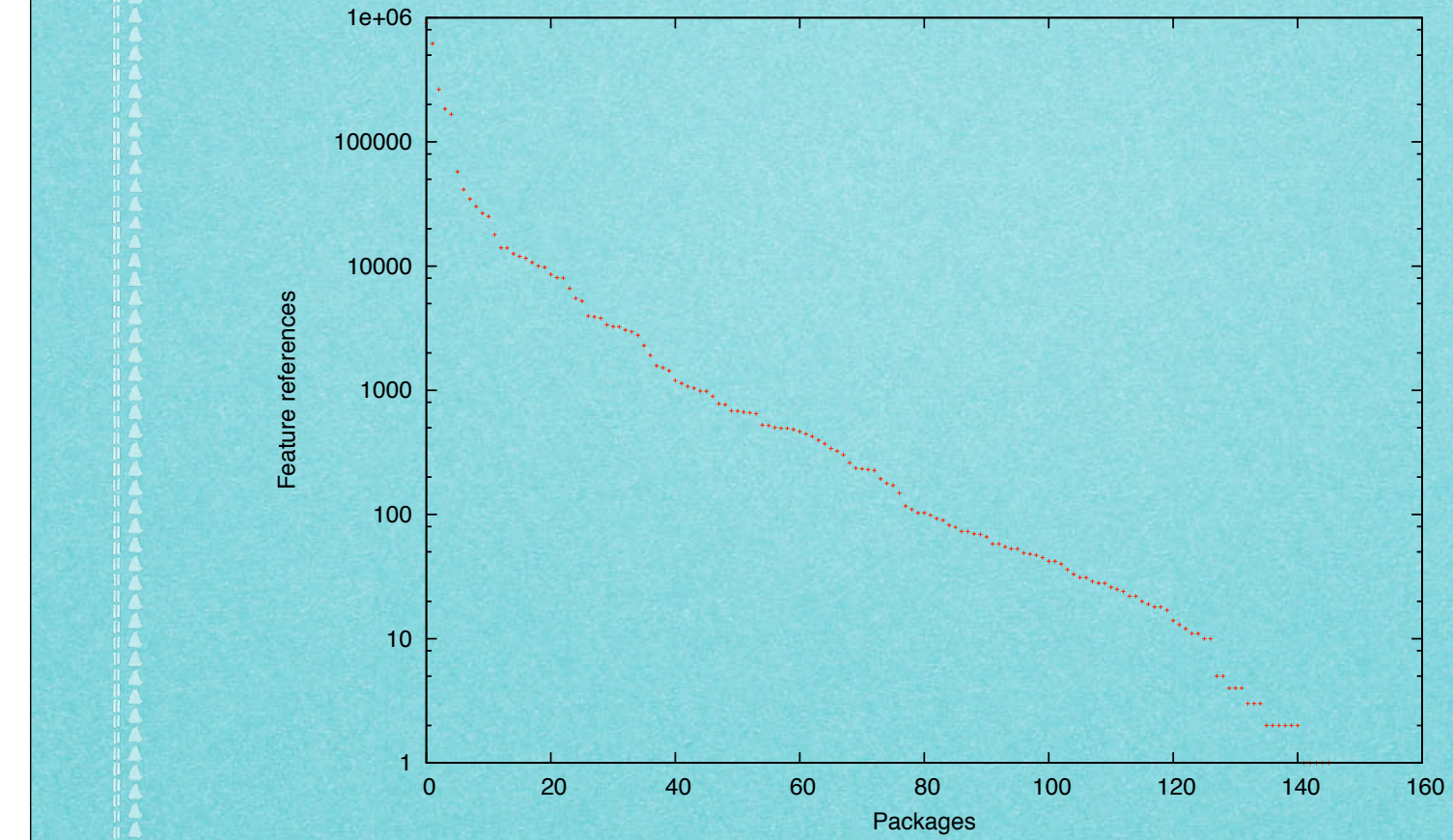
Partitioning of all method calls

(based on our SourceForge study)



API-usage frequency for Core APIs

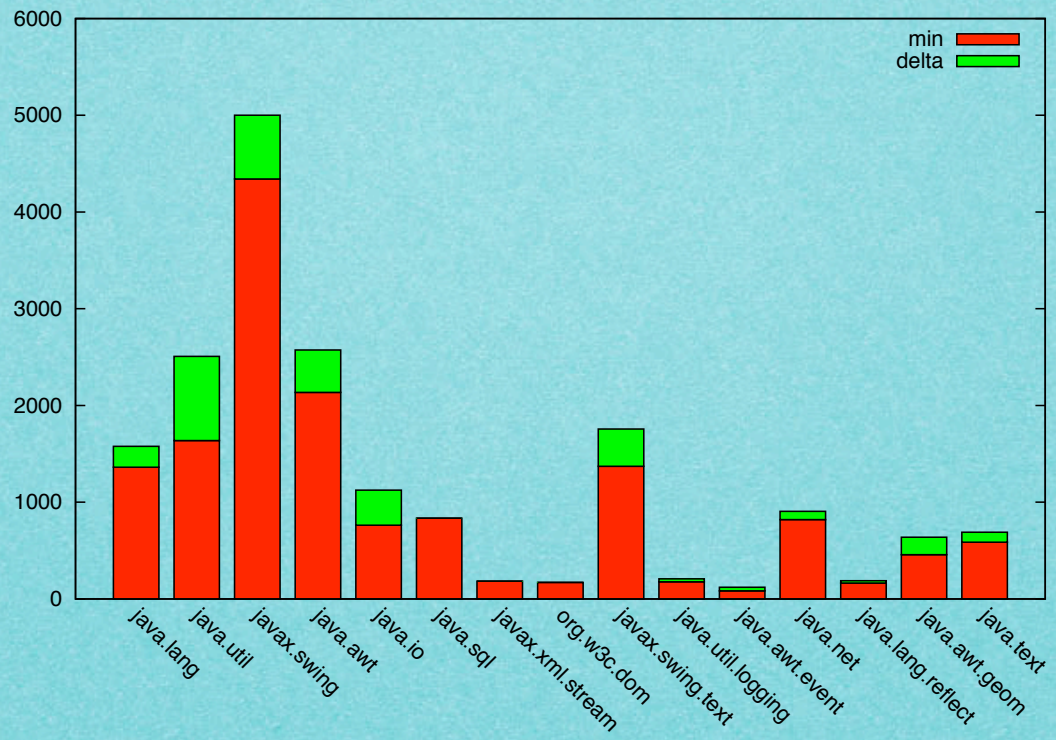
(based on our SourceForge study)



OverallAPIUsage/FRA_of_CorePackages_XYPlot.eps

API size min/max

(based on our SourceForge study)



Paths for API migration

- Manual migration using guides
- Refactoring-based
- **Wrapper-based re-implementation**
- Wrapping + Partial Evaluation
- Source-code transformation
- Byte-code transformation

The wrapper-based path to API migration

- Re-implement interface of *A* in terms of *B*.
- Use Adapter Design Pattern.

The wrapper-based path: Wrap jdom as dom

```
public class Node {
    protected org.jdom.Content content;
    ...
}

public class Element extends Node {
    ...
    public void appendChild(Node node) {
        org.jdom.Element elt = (org.jdom.Element)content;
        elt.addContent(node.content);
    }
    ...
}
```

The wrapper-based path: Wrap dom as jdom

```
public class Content {
    protected String text = null;
    protected org.w3c.dom.Node domNode = null;
    ...
    /*package*/ void build(Element parent) {
        domNode = parent.domDocument().createTextNode(text);
        parent.domElement().appendChild(domNode);
    }
}

public class Element extends Content {
    private String name = null;
    private List<Content> kids;
    ...
    public void addContent(Content elt) {
        kids.add(elt);
        if (domNode != null) {
            elt.build(this);
            domNode.appendChild(elt.domNode);
        }
    }
}
```

Virtual builder to turn deferred object into DOM object

Parenting leads to DOM tree construction

Discussion of wrapping

- Pros
  - *P* only needs a trivial if any transformation.
  - Existing test harness directly applies.
- Cons/Limitations
  - *A*'s interface continues to be used in *P*.
  - Hence, wrapping is not universally applicable.
  - Fully equivalent re-implementation is hard.
  - Possible inefficiency because of semantic gap.

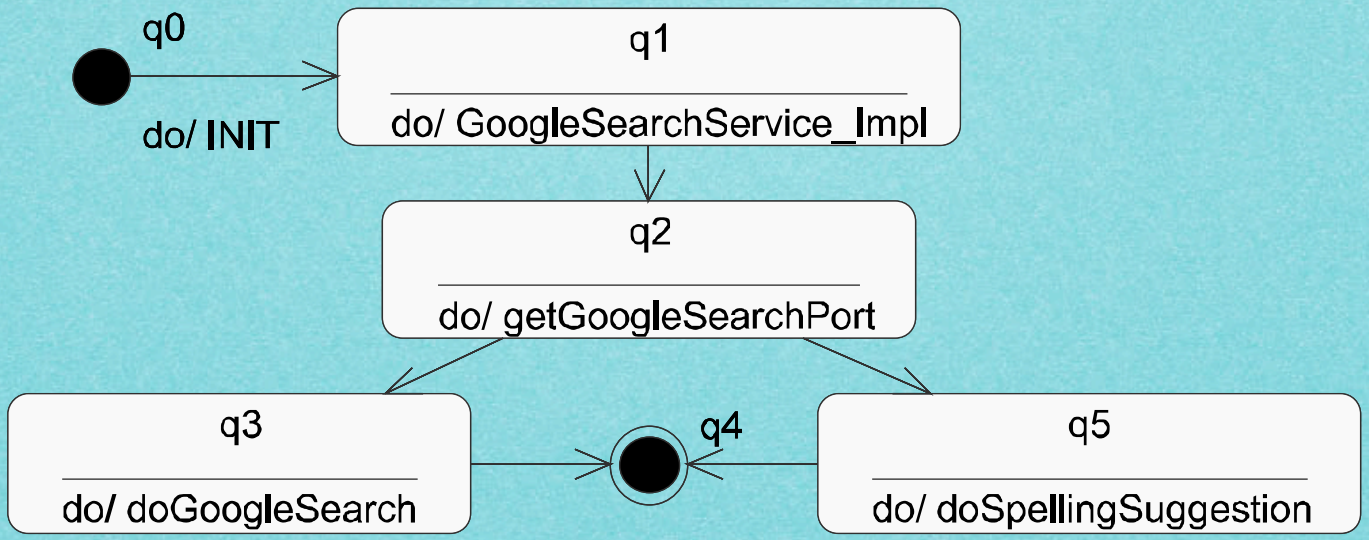
Wrapping with partial evaluation

- The path
  - Inline adapters and methods.
  - Perform constant propagation and friends.
- Challenges
  - Overall: readable and efficient code!
  - API-specific laws and analyses needed.
  - API usage scenarios need to be understood.

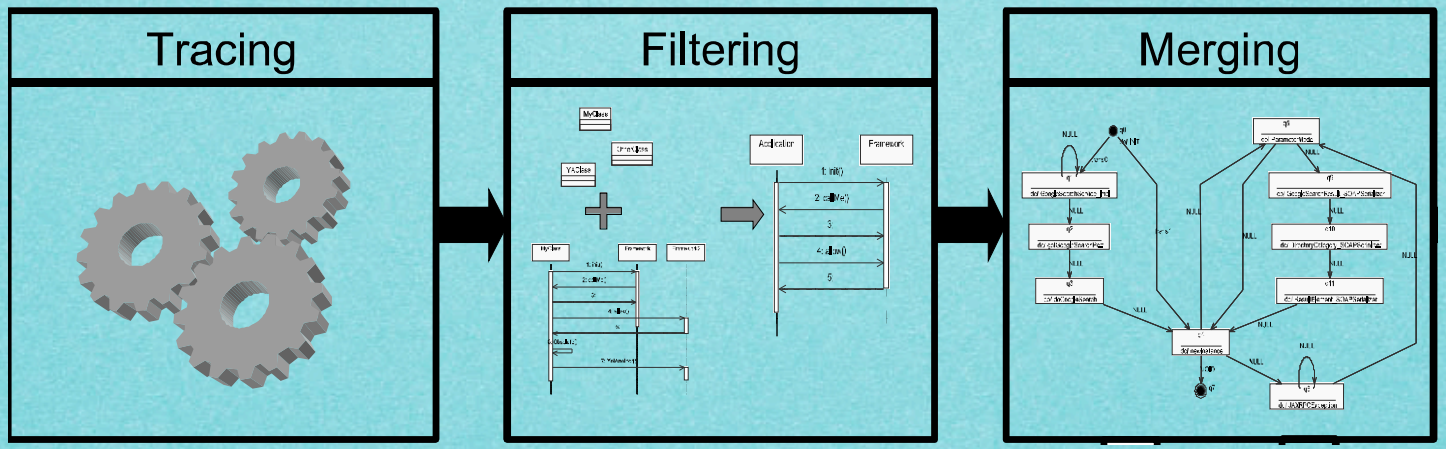
API protocols

- Some related work
  - Systä et al.'s API scenarios
  - Antkiewicz, Czarneck et al.'s FSMLs
- XML API scenarios
- Dynamic and static traces
- Context-free grammars as API protocols
- Attribute grammars as API protocols
- Outlook

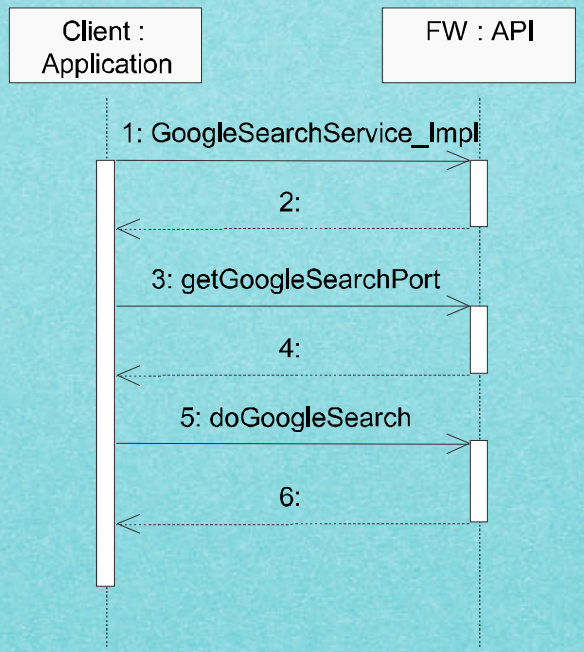
Systä et al.'s API scenarios



Systä et al.'s API scenarios



Systä et al.'s API scenarios



Antkiewicz, Czarnecki et al.'s Framework-Specific Modeling Languages

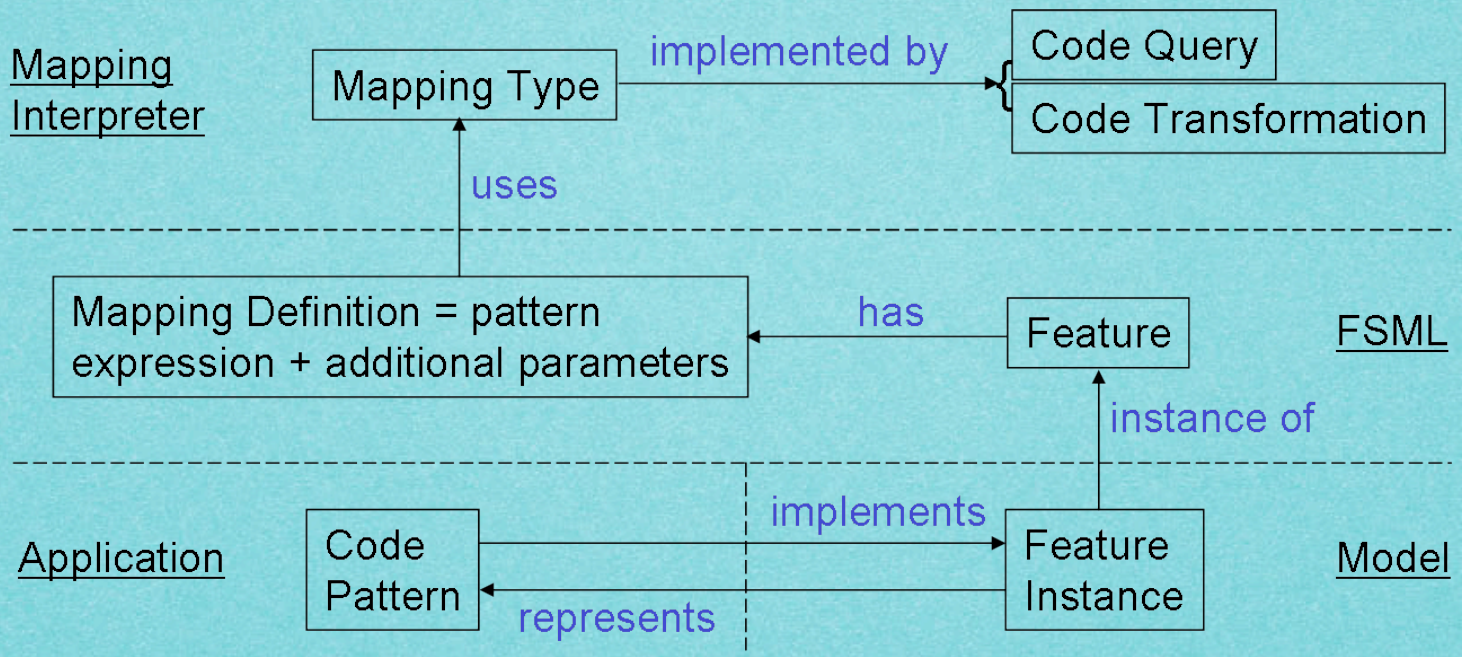
Feature hierarchy	***Explanation***
`Applet`	*concept (root feature)*
`  [1..1] name (String)`	*mandatory feature with attribute*
`  ![1..1] extendsApplet`	*essential feature*
`    [0..1] extendsJApplet`	*optional feature*
`  [1..1] lifecycleMethods`	*mandatory feature*
`   !<1-5>`	*essential* OR *feature group*
`      [0..1] init`	
`      [0..1] start`	
`      [0..1] paint`	*optional grouped features*
`      [0..1] stop`	
`      [0..1] destroy`	
`  [0..*] parameter`	*optional multiple feature*
`    [0..*] name (String)`	*multiple feature with attribute*
`  [0..1] providesParamInfo`	*optional feature*
`  [1..1] infoForParams`	*mandatory feature*

Antkiewicz, Czarnecki et al.'s Framework-Specific Modeling Languages

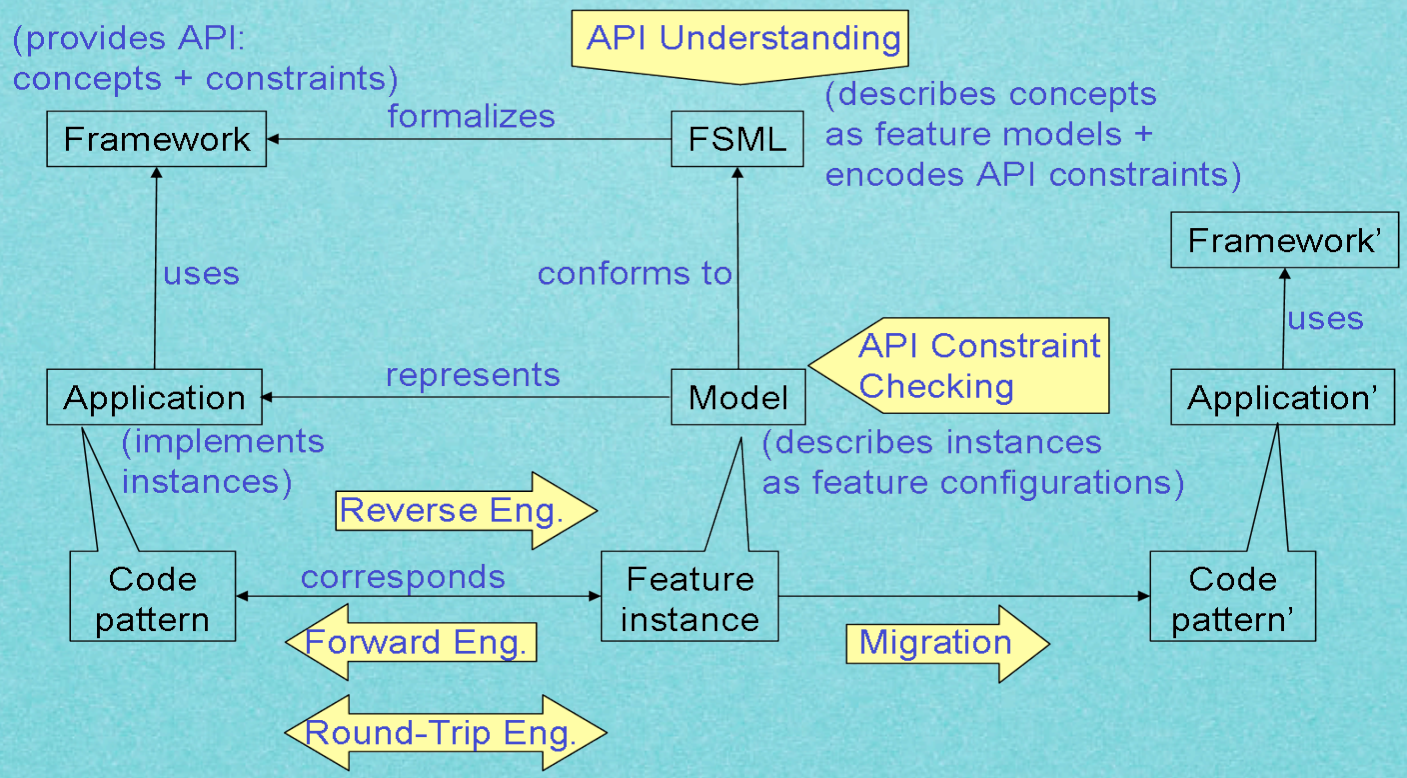
Feature hierarchy	***Feature correspondence explanation***
`Applet <class>`	*concept instance corresponds to a Java class*
`[1..1] name (String) <fullyQualifiedName>`	*value of the feature is a fully qualified name of the context class*
`![1..1] extendsApplet <assignableTo:'Applet'>`	*feature is present if the context class is a subtype of Applet*
`[0..1] extendsJApplet <assignableTo:'JApplet'>`	
`[1..1] lifecycleMethods`	*a feature used for grouping, no mapping definition*
`!<1-5>`	
`[0..1] init <methods:'void init()'>`	*each grouped feature corresponds to a non-inherited method of the context class*
`[0..1] start <methods:'void start()'>`	
`[0..1] paint <methods:'void paint(Graphics)'>`	
`[0..1] stop <methods:'void stop()'>`	
`[0..1] destroy <methods:'void destroy()'>`	

Mapping definition

Antkiewicz, Czarnecki et al.'s Framework-Specific Modeling Languages



Antkiewicz, Czarnecki et al.'s Framework-Specific Modeling Languages



XML programming scenarios

- *Visit* XML tree
- *Query* and *extract from* XML tree
- *Query* and *transform* XML tree
- *Build* XML tree

Remember our running example

Trace-based reasoning

- Trace = Execution of OO program
- Items in the trace:
  - Object constructions
  - Method calls
- Arguments and results are *object identities*.

Dynamic trace
(Construct Obama with jdom)

- **oid1 ← *new* Document()**
- **oid2 ← *new* Element(“contacts”)**
- **oid1.addContent(oid2)**
- **oid3 ← *new* Element(“person”)**
- **oid4 ← *new* Element(“name”)**
- **oid4.setText(“Barack Obama”)**
- **oid3.addContent(oid4)**
- **oid5 ← *new* Element(“age”)**
- **oid5.setText(“47”)**
- **oid3.addContent(oid5)**
- oid6 ← *new* Element(“person”)
- …

Static traces

- Analyse usage ...
  - without running the program,
  - while approximating all possible runs.
- Key ideas:
  - *Symbolic* object ids via *points-to analysis*.
  - Explicit representation of *iteration*.

Static trace
(Builder scenario with jdom)

- oid1 ← *new* Document()
- oid2 ← *new* Element(“contacts”)
- oid1.addContent(oid2)
- for all oid3:
  - oid3 ← *new* Element(“person”)
  - oid4 ← *new* Element(“name”)
  - oid4.setText(?)
  - oid3.addContent(oid4)
  - oid5 ← *new* Element(“age”)
  - oid5.setText(?)
  - oid3.addContent(oid5)

Context-free grammars as API protocols

- Language of traces
- Nonterminals = protocols
- Terminals = method or constructor calls

The context-free API protocol for recursive construction for the jdom API

- import org.jdom.*;
- **buildTree** = Document.*new*
  Element.*new*
  Document.addContent
  manyChildren
- **manyChildren** = (Element.*new*
  (Element.setText
  | **manyChildren**
)
  Element.addContent
)*

The context-free API protocol for recursive construction for W3C's dom API

- import org.w3c.dom.*;
- **buildTree** = DOMImplementation.createDocument
  Document.getDocumentElement
  manyChildren
- **manyChildren** = (Document.createElement
  ((Element.setText
  Document.createTextNode
  Element.appendChild
)
  | **manyChildren**
)
  Element.appendChild
)*

The skeleton of the protocols

- buildTree = manyChildren
- manyChildren = (ε | manyChildren)*

Beyond context-free protocols

- *Parameterization* to model object-id constraints
- *Wildcards* to model non-API actions
- *Interleaving* to relax order constraints
- Iterated matching to cover multiple scenarios
- Computational models
  - Attribute grammars
  - Logic programs
  - Process algebras
  - Graph transformation

Conclusion 1/2

- API migration - technical/conceptual challenges
  - Define an appropriate protocol notion.
  - Match protocols to verify protocol adherence.
  - Match protocols for API migration.
  - Fully leverage work on flow analysis.
  - Deal with semi-automatic status.
  - Preserve system testability.
  - ...

Conclusion 2/2

- API migration - high-level challenges
  - Show that general migration is feasible.
  - Make migration technically manageable.
  - Make migration economically viable.