Reverse Engineering Java Using ASF+SDF and Rigi

A preliminary experience report

Eva van Emden, CWI

Contents

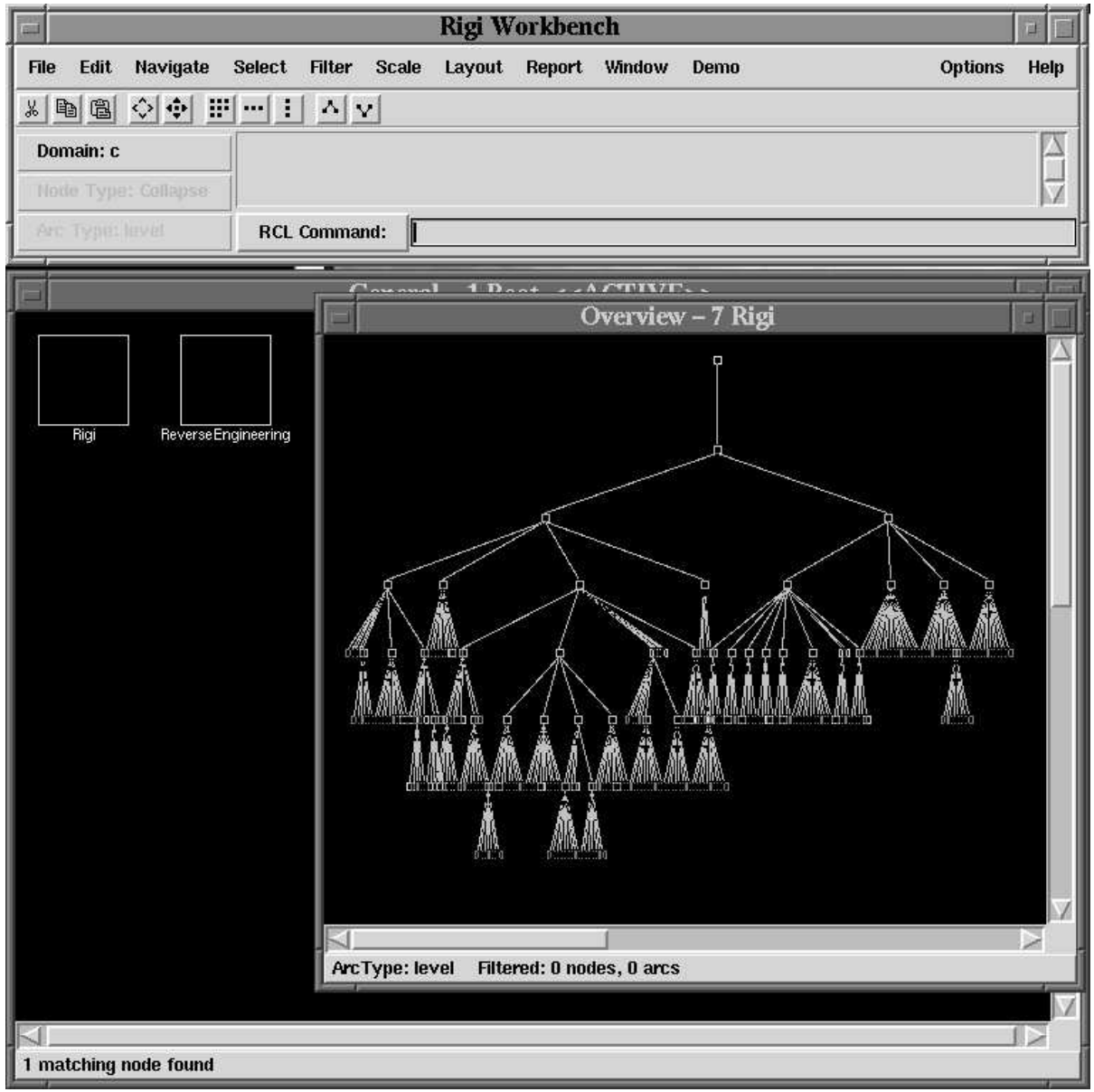
- Introduction to Rigi and the Rsf format
- Java2RSF: Translating with an ASF+SDF specification
- Visualizing Java code smells in Rigi

What is Rigi?

- Visual reverse engineering tool
- Rigi represents a program as a collection of nodes and arcs
  - Nodes represent features in the program, such as methods or classes
  - Arcs represent relationships between the nodes, such as “contain” or “call”
- Different views can be created by:
  - Filtering out certain node and arc types
  - Using the built-in layout algorithms
  - Writing scripts in the Rigi command language (RCL)

Rigi Screenshot

A subsystem hierarchy view for a C program



Rigi standard format (RSF)

- Interchange format between parsing and processing
- Graph description language
- Simple text file
- Each line describes a node, arc, or attribute
- There are tools to translate between RSF and the GXL Graph Exchange Format

RSF

```
rsf-file:              <rsf-tuples>
rsf-tuples:            <rsf-tuple> "/n"
                       <rsf-tuples>
rsf-tuple:             <node-definition>
                       <arc-definition>
                       <attribute-definition>
node-definition:       "type" <node-spec> <node-type>
arc-definition:        <arc-type> <node-spec> <node-spec>
attribute-definition: <attribute-type> <node-spec>
                           <attribute-value>
node-spec:             <identifier>
node-type:             <identifier>
arc-type:              <identifier>
attribute-type:        <identifier>
attribute-value:       <identifier>
```

Structured RSF

- Each node has a unique number
- The file must start with a root and there must be "level" arcs connecting the root node to every node in the top level of the graph

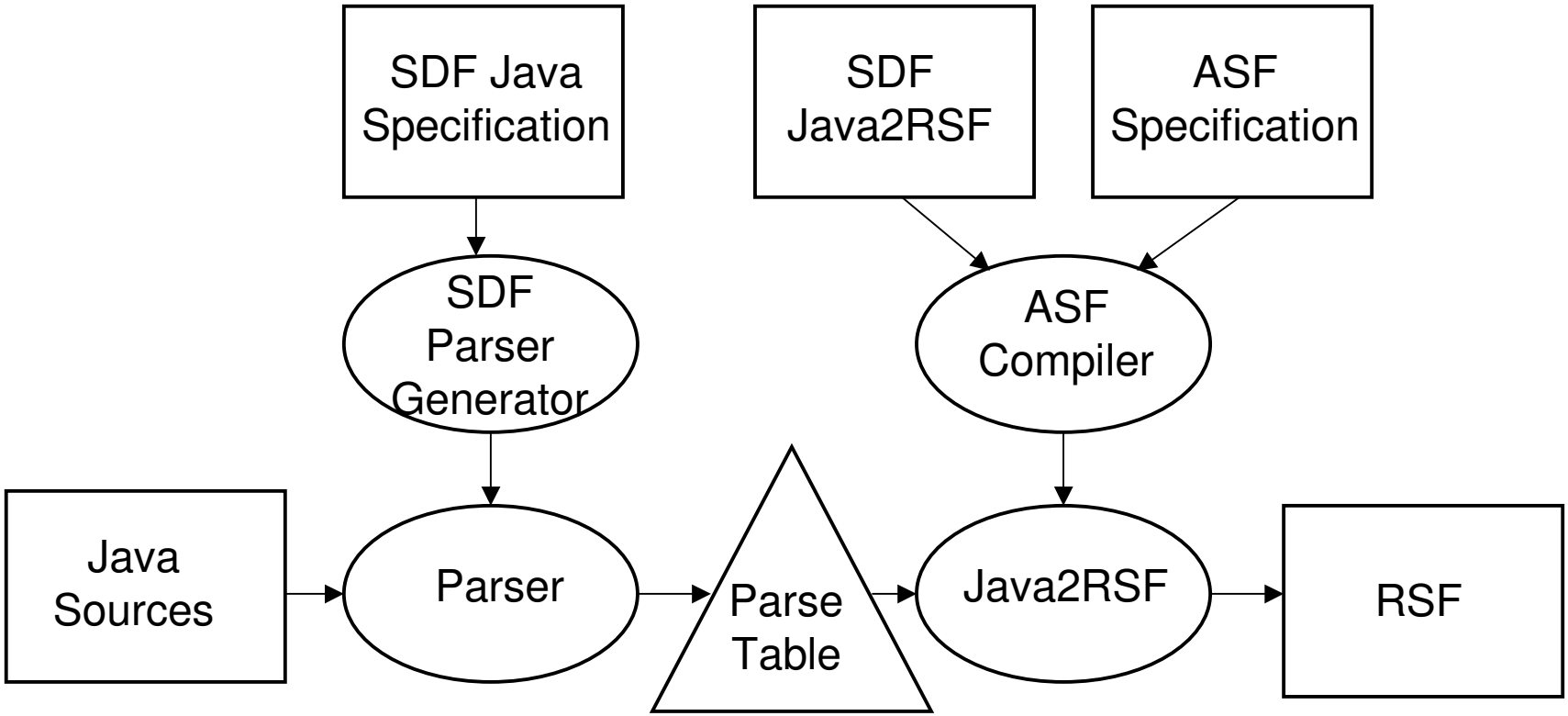
Rigi Domains

- A collection of node, arc, and attribute types used to describe a particular language
- Specified by creating a new directory with the name of the domain in the Rigi domain directory and adding text files specifying valid node, arc, and attribute types

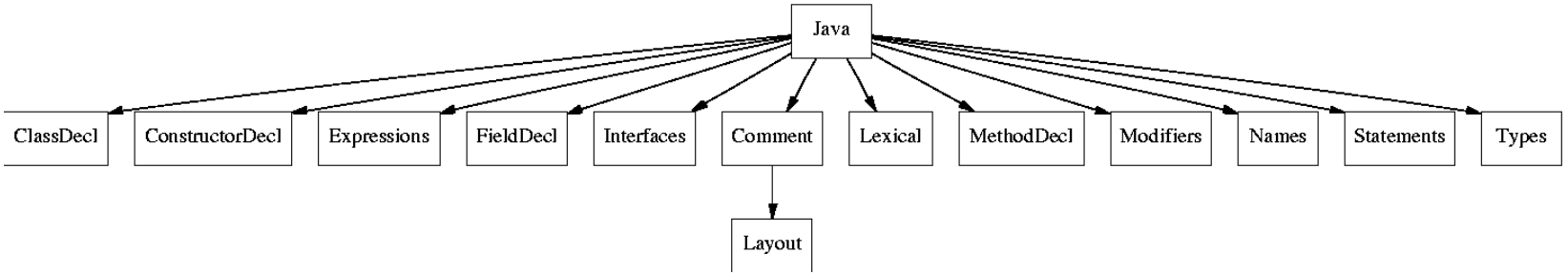
The Rigi Java Domain

- Nodes: Package, Class, Interface, Method, Constructor, Variable etc.
- Arcs: contain, call, access, isSuper, implementedBy etc.
- Attributes: visibility, static, abstract etc.

Java to RSF Translation

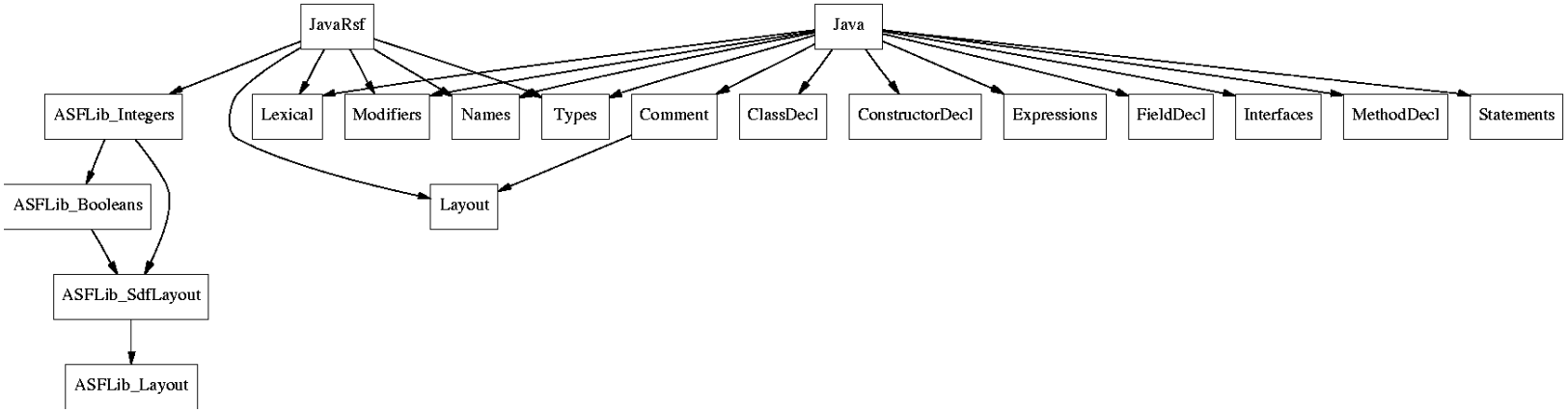


SDF Java Specification



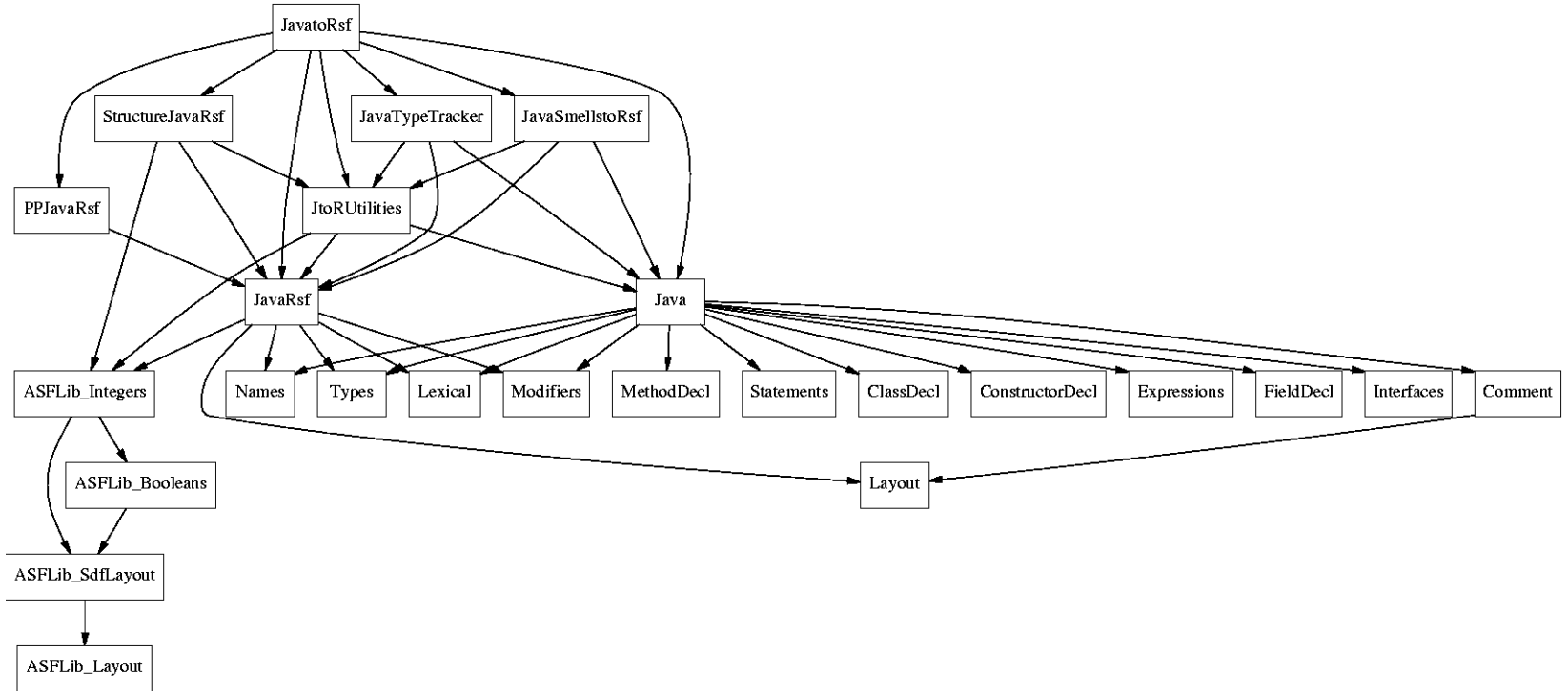
- Java grammar taken from online grammar base
- Some modifications made before it parsed all input files successfully

Java RSF Specification



- Describes RSF in the Java domain
- Makes use of standard ASF library components
- Refers to certain Java modules

Specification of Full Translator



Rewriting

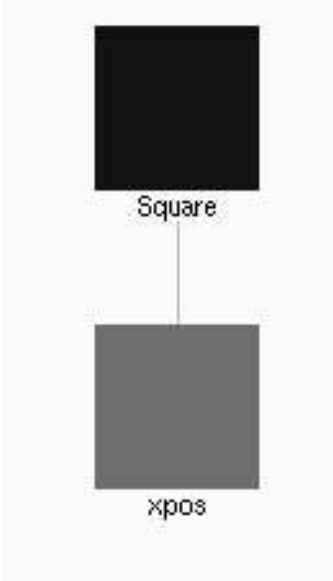
- Recognize certain Java constructs and output corresponding RSF

e.g.

```
public class Square {
   public Position xpos;
}
```

becomes

```
type Square Class
type xpos Variable
contain Square xpos
```



Rewriting: Traversal Functions

Function signature in SDF specification:

```
methodinv(Block, RsfTuple*, Name, Name) -> RsfTuple*
      {traversal(accu,  bottom-up)}

methodinv(MethodInvocation, RsfTuple*, Name, Name) ->
      RsfTuple* {traversal(accu, bottom-up)}
```

Rewriting: Traversal Functions (2)

`methodinv` traversal function is called in ASF:

```
[mb1]
Methodbdy(_Block,_RsfTuple*,_MethodName,_ClassId) =

_RsfTuple*
methodinv(_Block, , _MethodName, _ClassId)
```

Rewriting: Traversal Functions (3)

Rewrite rule for a `methodinv` match:

```
[mi1] _Type=getType(_Identifier0),
      _MethodName2 = _Type._Identifier1
==========================================
methodinv(_Identifier0._Identifier1(_ExpressionList*),
_RsfTuple*,_MethodName1,_ClassId) =

_RsfTuple*
call _MethodName1 _MethodName2
```

The Power of Traversal Functions

- Consider how many possibilities there are for a method invocation to appear in a statement:
  - s.draw();
  - if (s.isBlue()){…};
  - current = (Shape)list.getNext();
  - java.lang.Math.max(s.getx(), s.gety());
- Very tedious and error-prone to write rules to match all of these possibilities by hand

Using Rigi to Provide Refactoring Support

- Test system of 60 000+ loc
- System is being refactored to improve maintainability
- Decided to display code smells to see if visualizing them could be useful
- What are code smells?
  - A code smell is a symptom that may indicate something wrong in the code (Beck and Fowler)
  - A clustering of a code smells visible in Rigi may indicate a class or package that needs to be refactored

Visualization Options

- colour nodes according to degree of smell present (i.e. red smells, green does not), but this can't be done in rigi
- Each instance of a smell appears as a node attached to the method or class
- Smells currently implemented:
  - Typecasts
  - Instanceof
  - Switch statements

Smell Detection: ASF+SDF

- New smell detection module added to ASF+SDF specification

SDF:

```
smell(Block, RsfTuple*, Name, Name)
        -> RsfTuple* {traversal(accu,bottom-up)}
smell(Expression, RsfTuple*, Name, Name)
        -> RsfTuple* {traversal(accu,bottom-up)}
```

ASF:

```
[s2] smell(_Expression instanceof _ReferenceType,
   _RsfTuple*,_MethodName, _ClassId) =
_RsfTuple*
type _NodeSpec Instanceof
contain _MethodName _NodeSpec
```

Smell Detection: RSF

- Now a problem shows up: if method “draw” in the Java code contains two instanceofs, we get the following RSF:

```
type instanceof Instanceof
type instanceof Instanceof
contain draw instanceof
contain draw instanceof
```

Solution: Adding Structure to the RSF

- Standard RSF deletes all duplicate lines and therefore cannot have two nodes with the same name
- To allow multiple smell nodes to show I had to switch to producing partially structured RSF

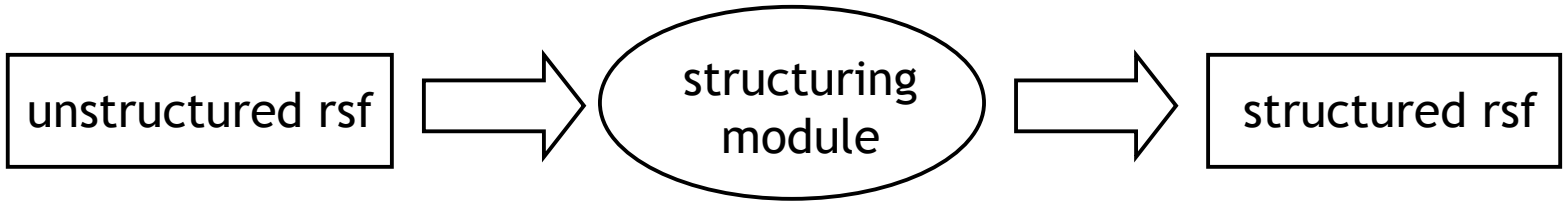
```
type instanceof Instanceof
type instanceof Instanceof
contain draw instanceof
contain draw instanceof
```

Becomes ⟹

```
type 1!Root Unknown
type 2!draw Method
level 1!Root 2!draw
type 3!instanceof Instanceof
level 1!Root 3!instanceof
type 4!instanceof Instanceof
level 1!Root 4!instanceof
contain 2!draw 3!instanceof
contain 2!draw 4!instanceof
```

Smell Detection: Adding Structure to the RSF

- Add a structuring module to the existing specification



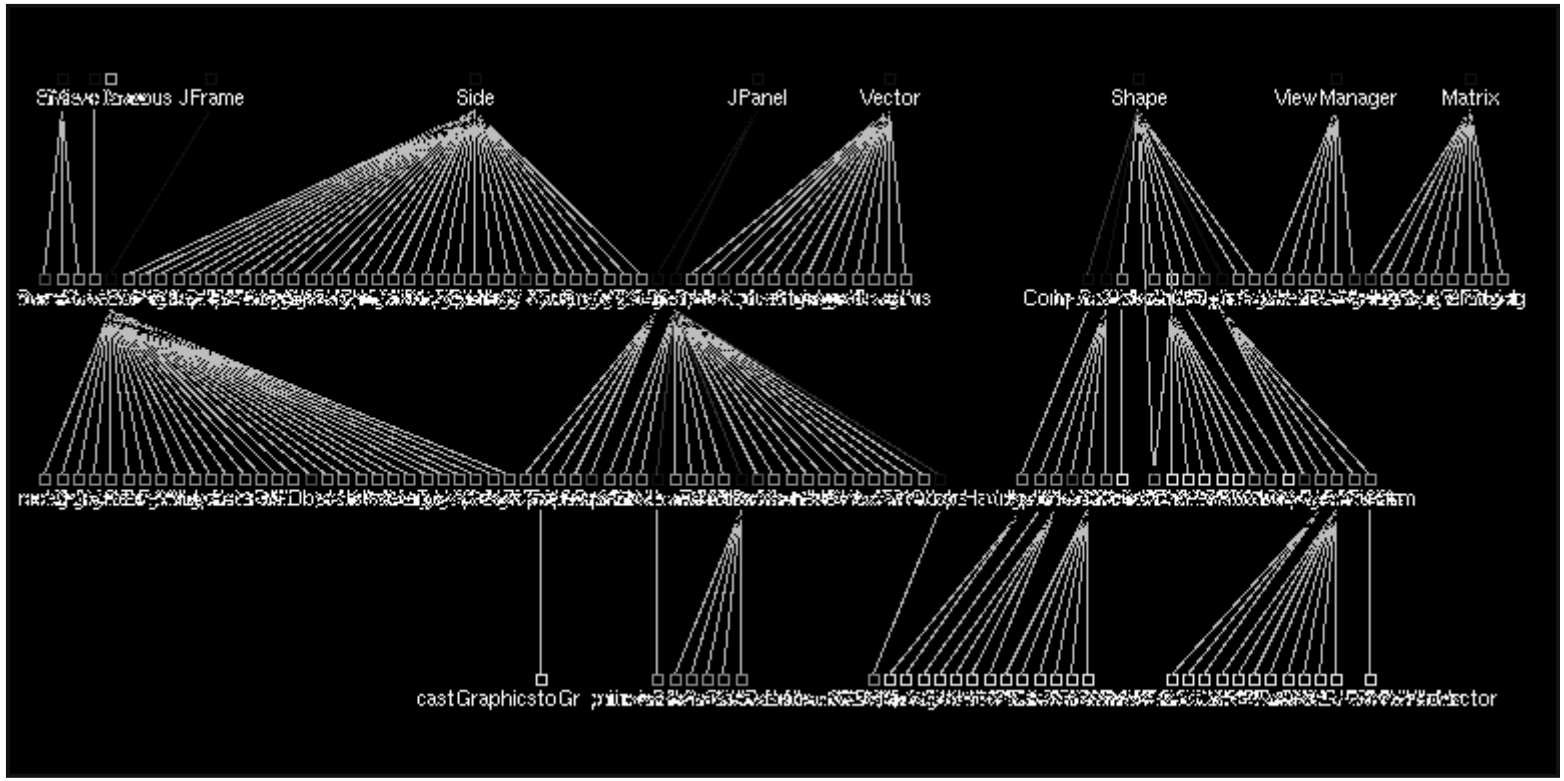
- Structuring process:
  - Unique all the rsf tuples
  - Take all the node names and assign them unique node numbers
  - Replace all node names with the numbered version
  - Place a level tuple after each node definition

Smell Detection: Rigi Display

- Add smell node types to the Java domain in Rigi
- Write a script in the Rigi command language to produce a meaningful view in Rigi

Visualizing Code Smells

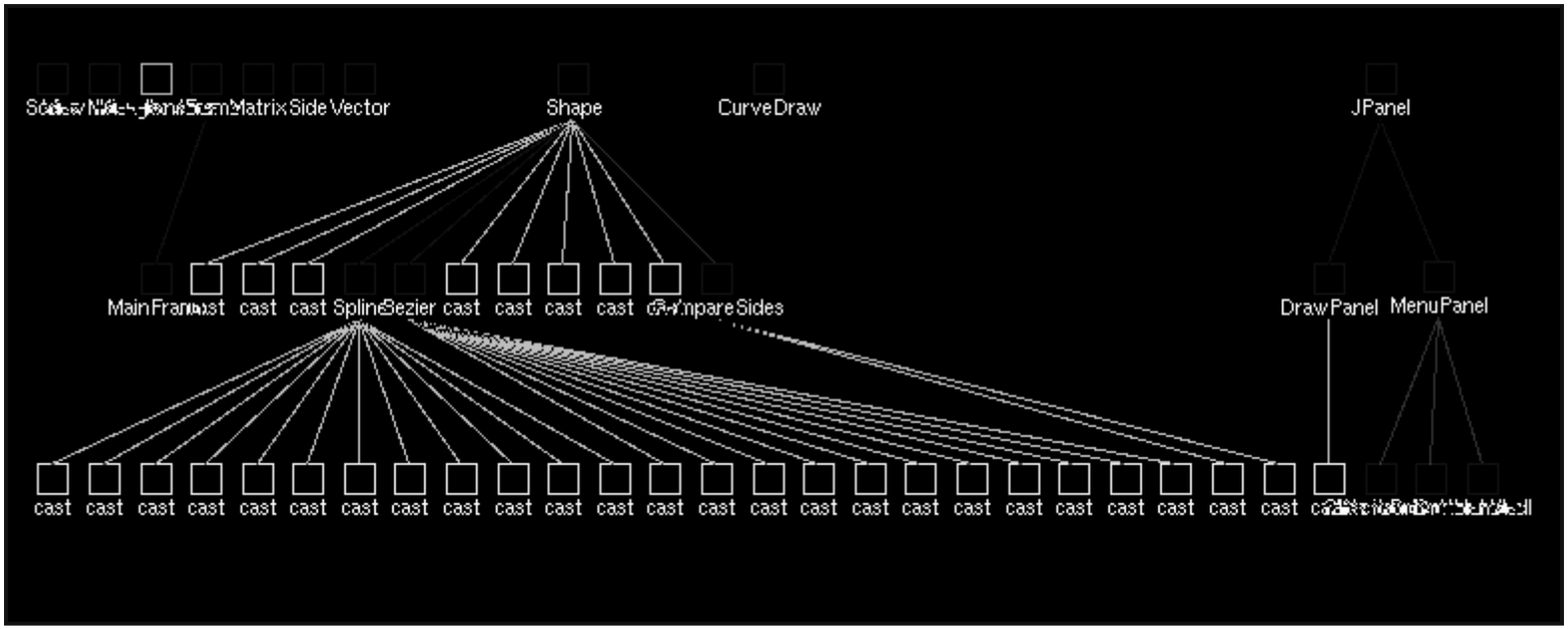
Rigi View 1



All nodes except classes, methods, constructors and typecasts have been filtered out and a layout algorithm applied.

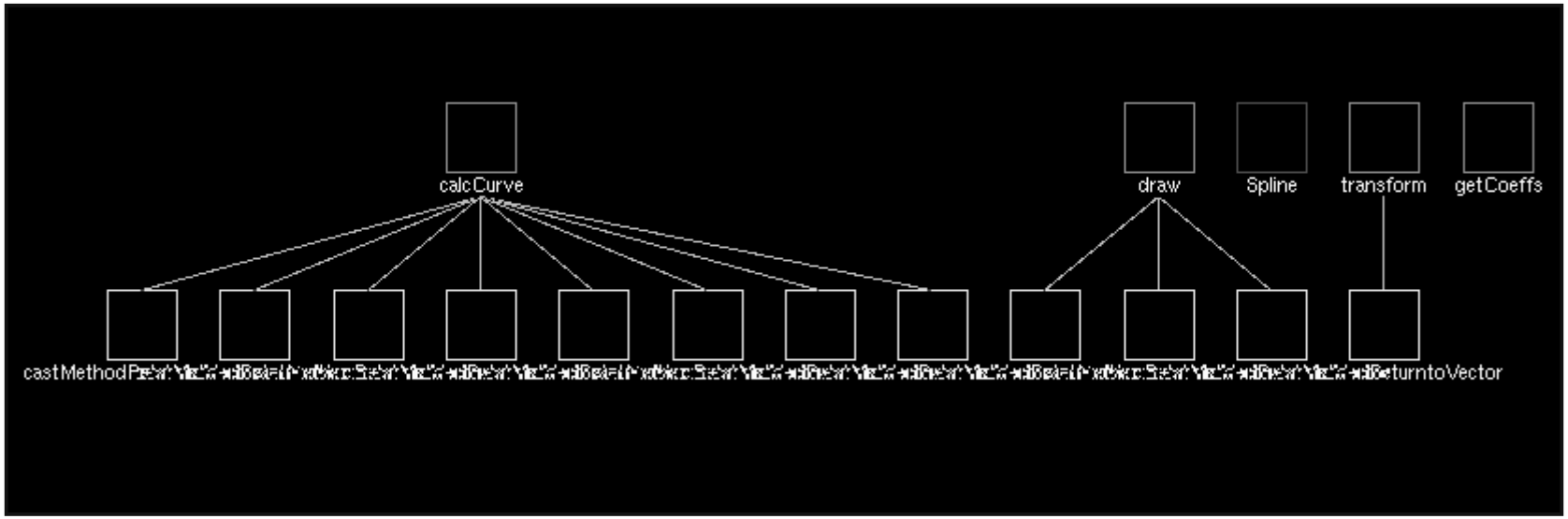
Eva van Emden, CWI

Rigi View 2: Show Smell By Class



- methods collapsed into their classes
- All the casts inside a class are attached to that class…

Rigi View 2: Show Smell By Class (2)



- …but a class node can be opened to show the members inside with their cast nodes attached

Where to Go From Here?

- Continue to experiment with views
- Expand to displaying further code smells
- Find a way to make the specification more efficient
  - Small programs (several kloc) ok
  - Does not finish in reasonable time (at all) on our 60 kloc test system
- Finish making the specification correct and complete
  - Still some problems with getting types to show method calls properly

Eva van Emden, CWI