On the use of graph transformations for **model refactoring**

Tom Mens

Software Engineering Lab
University of Mons-Hainaut
http://w3.umh.ac.be/genlog

Tutorial outline

✓Introduction

✓Graph transformation theory

- Graph transformation experiments
- Conclusion

Graph Transformation Experiments

GT Experiments

- Introduction
- GT theory
- Experiments
- Conclusion

Goal: provide graph transformation support for model refactoring

dealing with conflicts and dependencies between refactorings
- In AGG
- Based on critical pair analysis

generating refactoring code from graph transformations
1. In Fujaba ...
2. ... or CASE-tool independent

proving that refactorings preserve certain behavioural properties
- Formal approach

GT Experiments

Comparison of graph transformation and refactoring concepts

graph transformation	refactoring
type graph and global graph constraints	well-formedness constraints
negative application conditions	refactoring preconditions
graph production	refactoring transformation
programmed GT	composite refactoring
critical pair analysis and parallel dependence	detecting refactoring conflicts
sequential dependence	causal dependencies between refactorings

Refactoring dependencies

Experiment in AGG:
using critical pair analysis

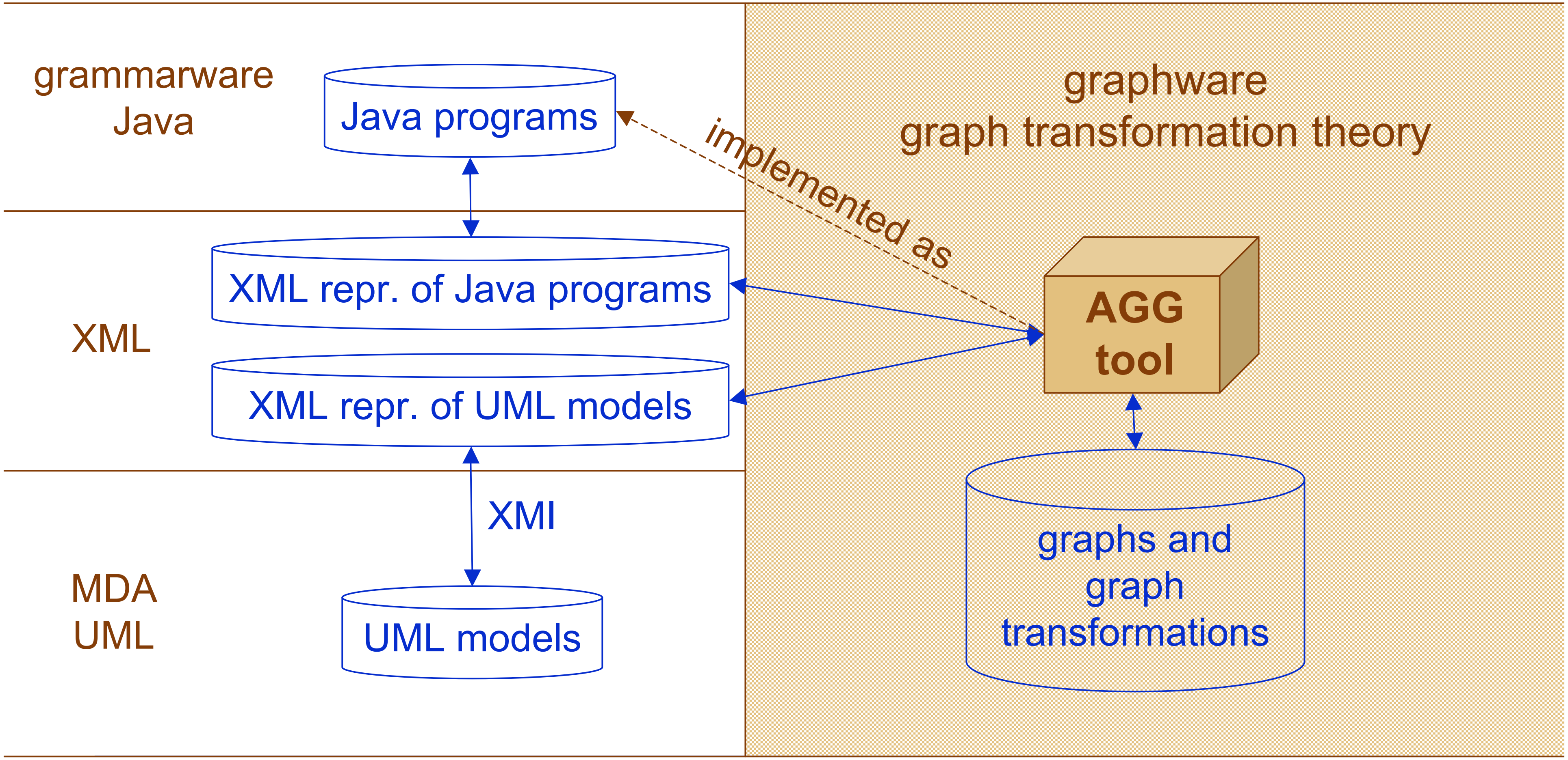
(in collaboration with *Gabriele Taentzer*,
Technical University of Berlin)

Refactoring dependencies

- About *AGG* (Attributed Graph Grammar system)
  - Algebraic approach to graph transformation
  - Annotations are in Java
  - Efficient graph parsing
    - Parse grammar
    - Critical pair analysis
  - Easy integration with Java code

Refactoring dependencies

- About *AGG* : Technological Spaces



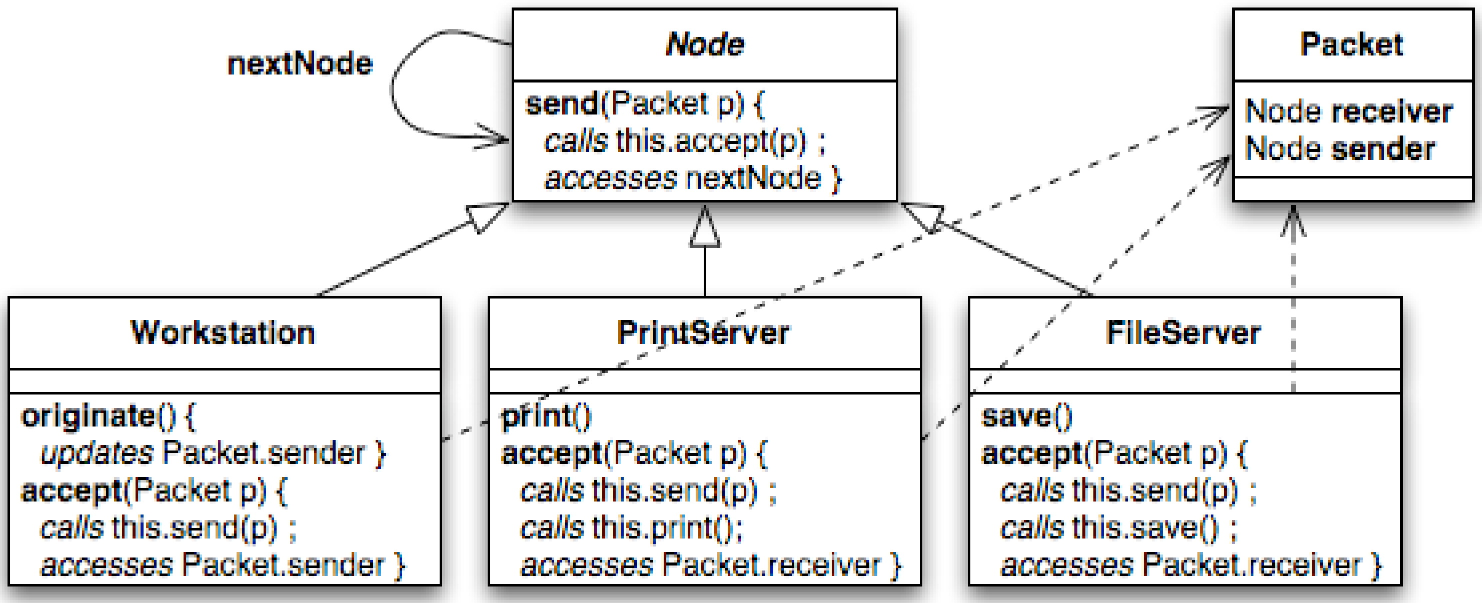
Refactoring dependencies

- Concrete Scenario: Suggest refactoring opportunities
  - What are the alternatives of a selected refactoring?
  - Which other refactorings need to be applied first in order to make the selected refactoring applicable?
  - Which other refactorings are still applicable after applying the selected refactoring?

- Goal: Automate the detection of
  - mutual exclusion relationships between refactorings
  - sequential dependencies between refactorings

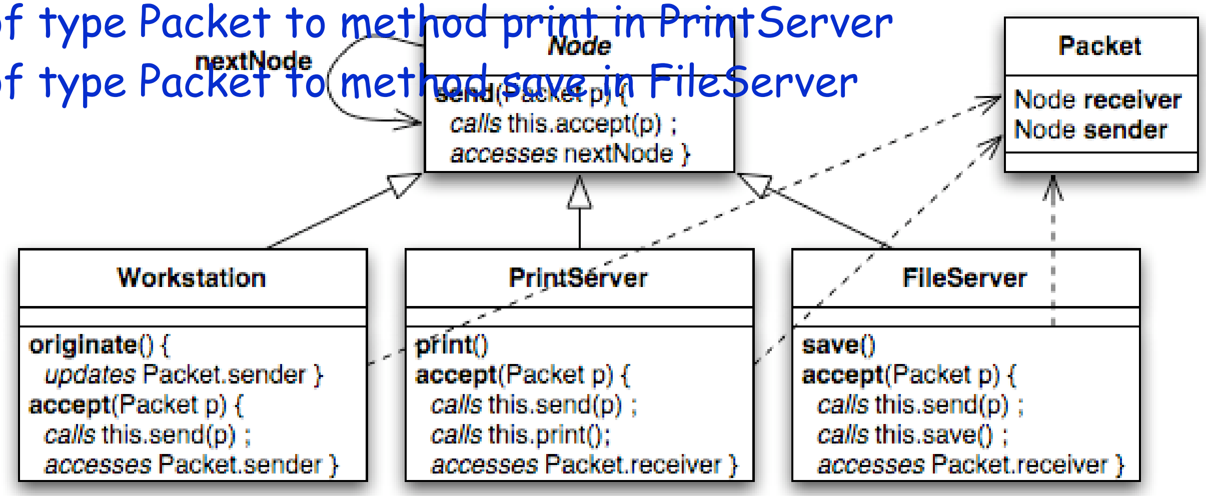
Refactoring dependencies

- Example



Refactoring dependencies

- Refactoring opportunities
  - T1 Rename Method print in PrintServer to process
  - T2 Rename Method save in FileServer to process
  - T3 Create Superclass Server for PrintServer and FileServer
  - T4 Pull Up Method accept from PrintServer and FileServer to Server
  - T5 Move Method accept from PrintServer to Packet
  - T6 Move Method accept from FileServer to Packet
  - T7 Encapsulate Variable receiver in Packet
  - T8 Add Parameter p of type Packet to method print in PrintServer
  - T9 Add Parameter p of type Packet to method save in FileServer



Refactoring dependencies

	T1	T2	T3	T4	T5	T6	T7	T8	T9
T1	×								
T2		×							
T3			×						
T4				×	×	×			
T5					×	×			
T6						×		×	×
T7							×		
T8								×	×
T9									×

× critical pairs
(negative sequential dependencies)

Refactoring dependencies

	T1	T2	T3	T4	T5	T6	T7	T8	T9
T1	×							>>	
T2		×							>>
T3			×						
T4				×	×	×			
T5					×	×			
T6						×		×	×
T7							×		
T8								×	×
T9									×

>> critical pairs that can be serialised

Refactoring dependencies

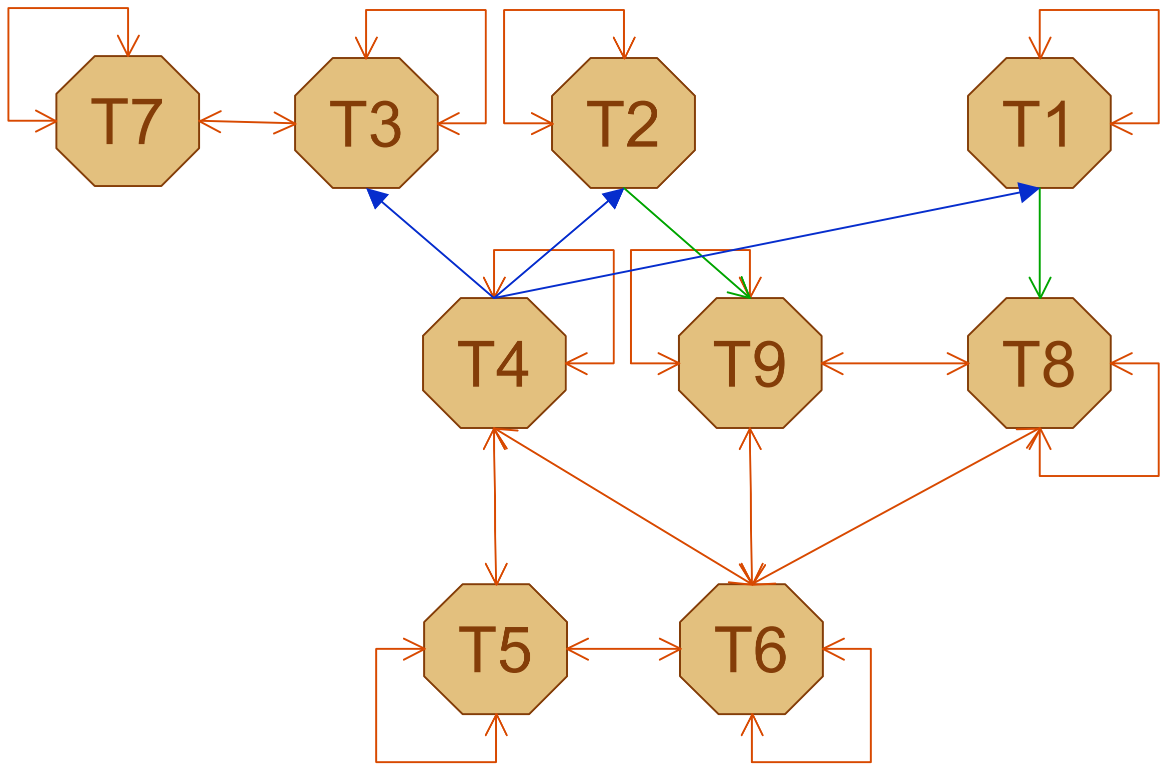
	T1	T2	T3	T4	T5	T6	T7	T8	T9
T1	×			←					
T2		×		←					>>
T3			×	←					
T4				×	×	×			
T5					×	×			
T6						×		×	×
T7							×		
T8								×	×
T9									×

rename to *process* needed before pull up of *accept* can be done

← positive sequential dependencies

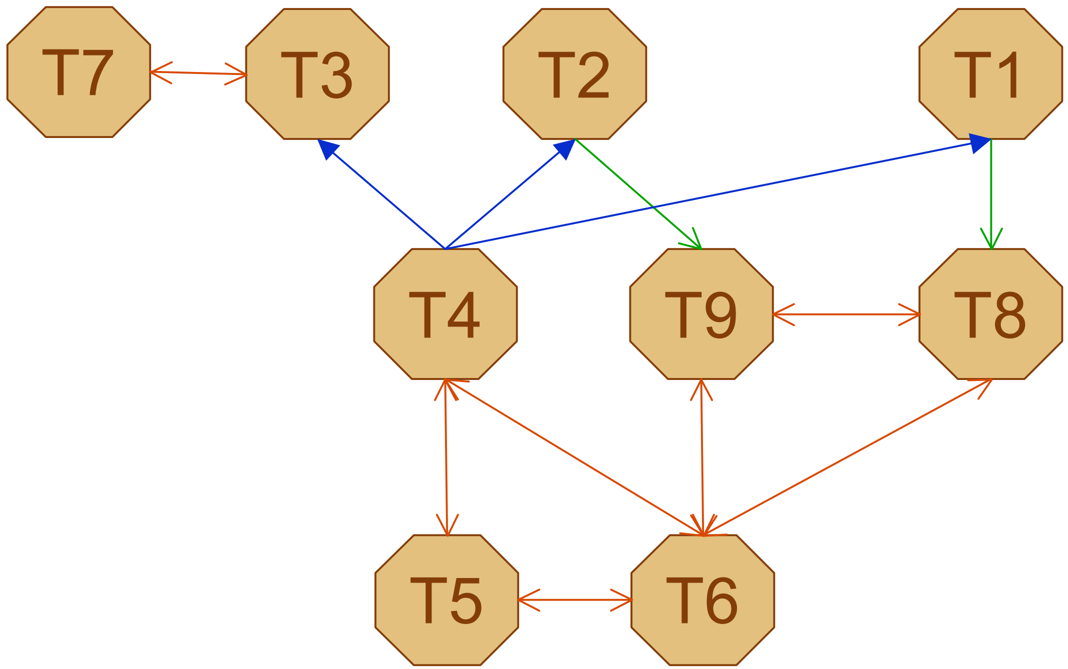
Refactoring dependencies

- Dependency graph



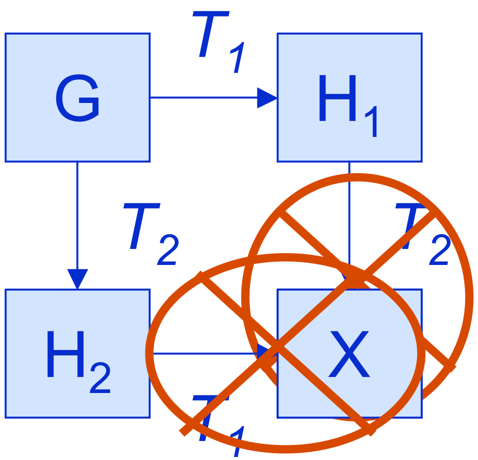
Refactoring dependencies

- Dependency graph (without self-cycles)



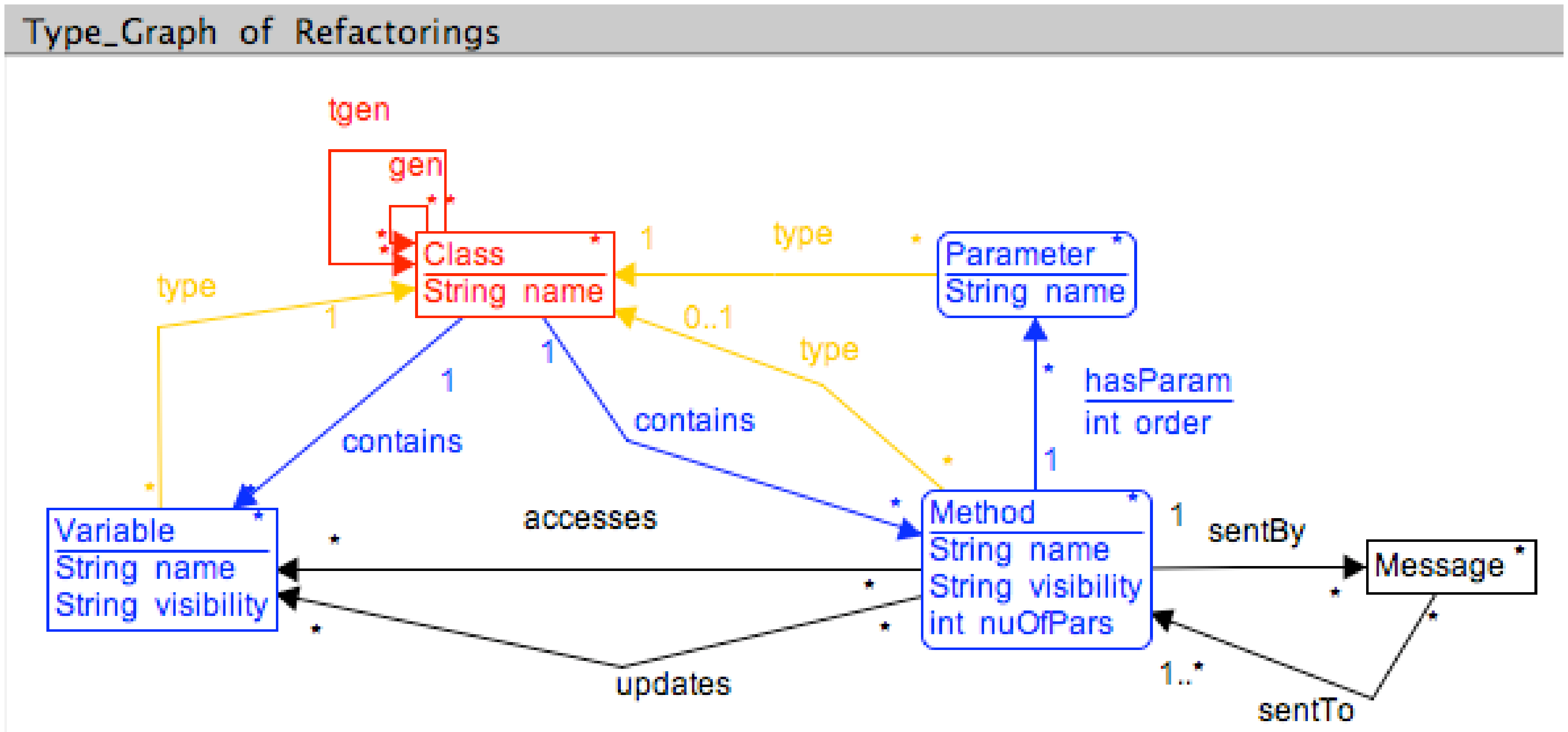
Refactoring dependencies

- Approach: Use critical pair analysis in AGG
  - T_1 and T_2 form a *critical pair* if
    - they can both be applied to the same initial graph G but
    - applying T_1 prohibits application of T_2 and/or vice versa



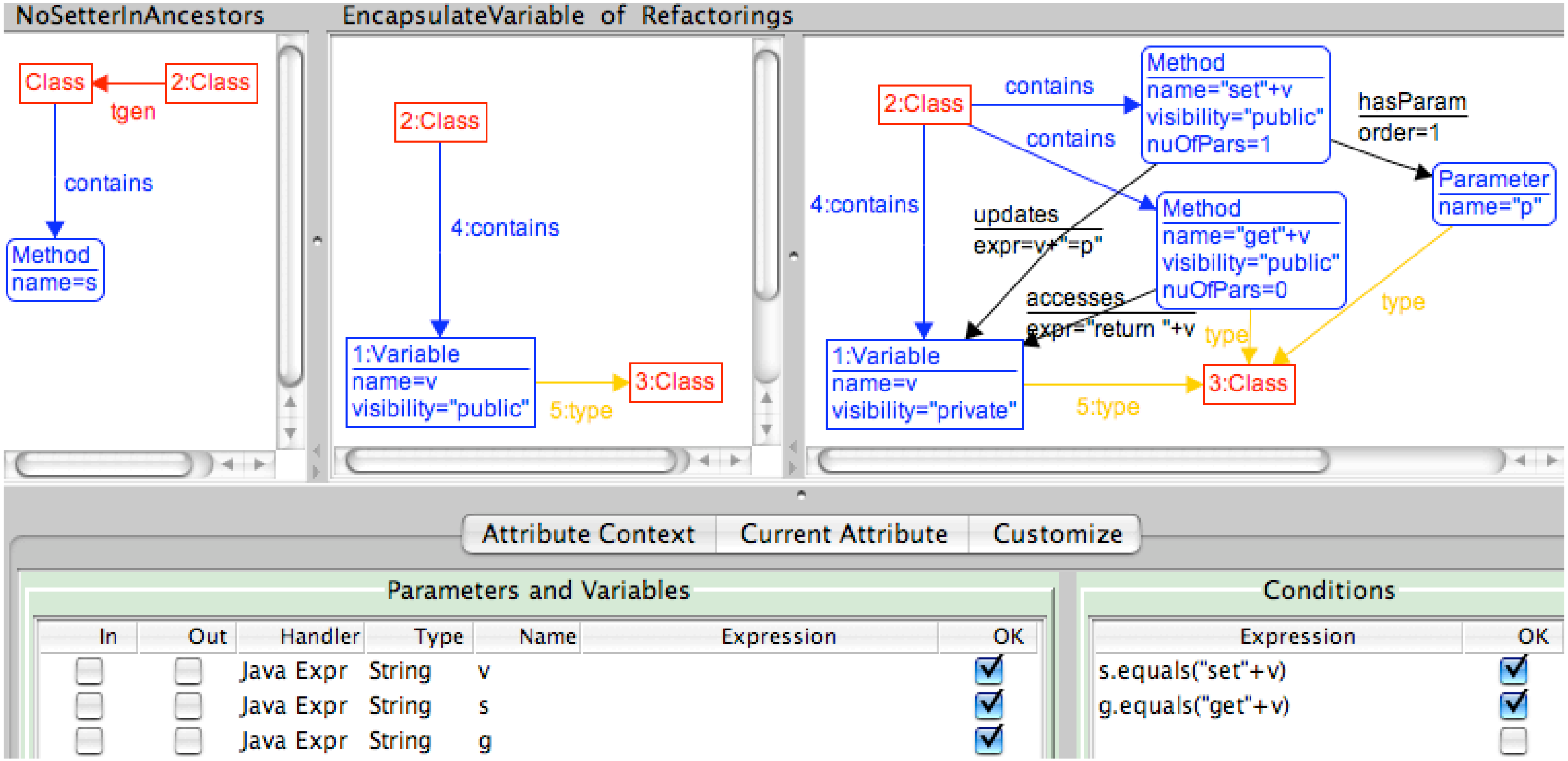
Refactoring dependencies

Step 1: Express object-oriented metamodel as (attributed) type graph



Refactoring dependencies

Step 2: Express refactorings as (typed attributed) graph transformations



Refactoring dependencies

Step 3: Statically detect *critical pairs* between refactoring transformations

- *Potential conflicts* between refactorings

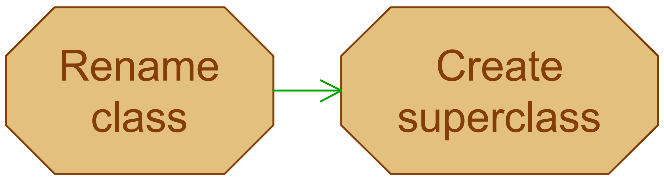
Critical Pairs

first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	2	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	2	0	0	0	0	0	0	1
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	1	1	1	0	0	2

Refactoring dependencies

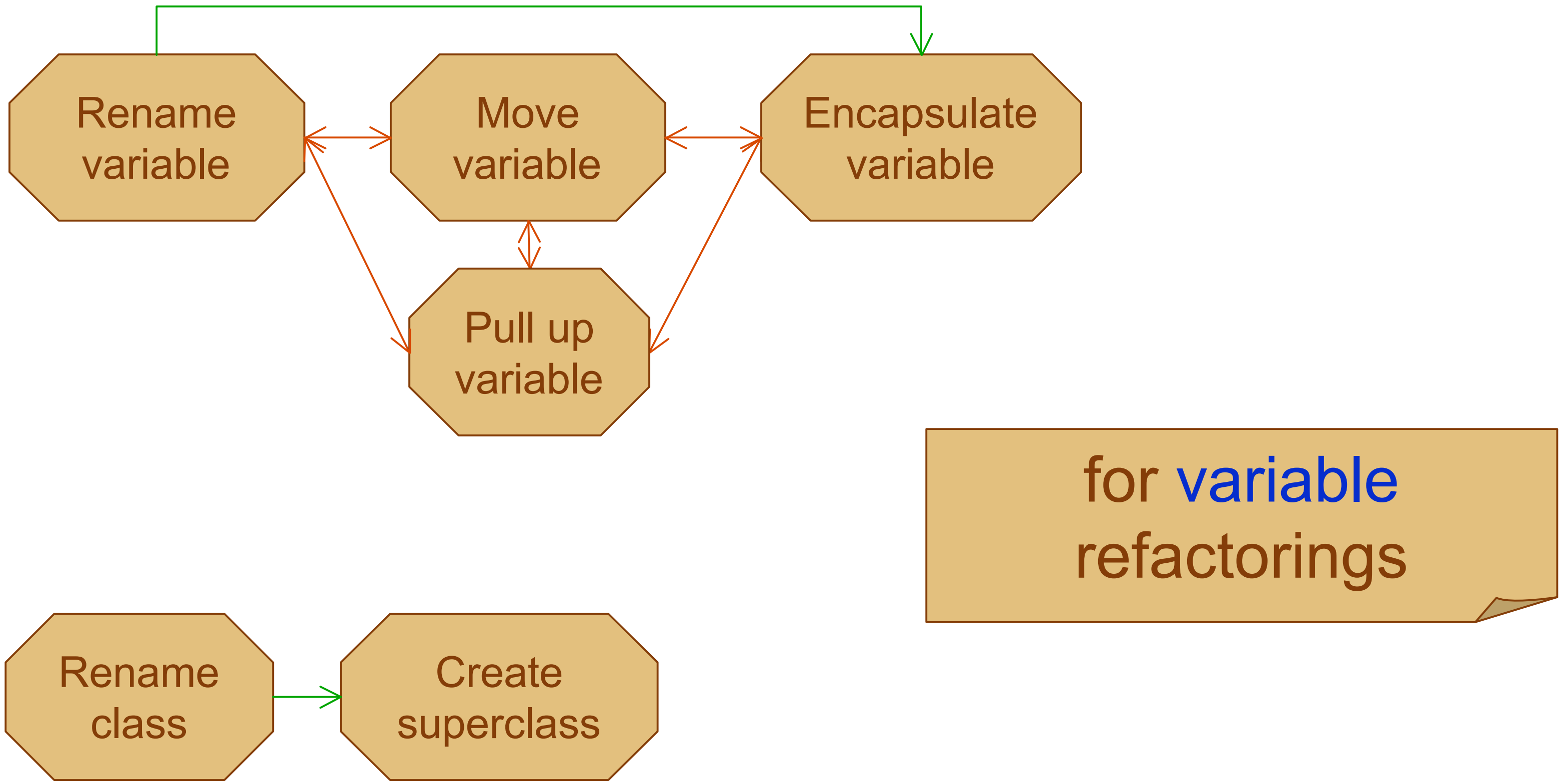
Step 4: Analyse dependency graph

for class refactorings



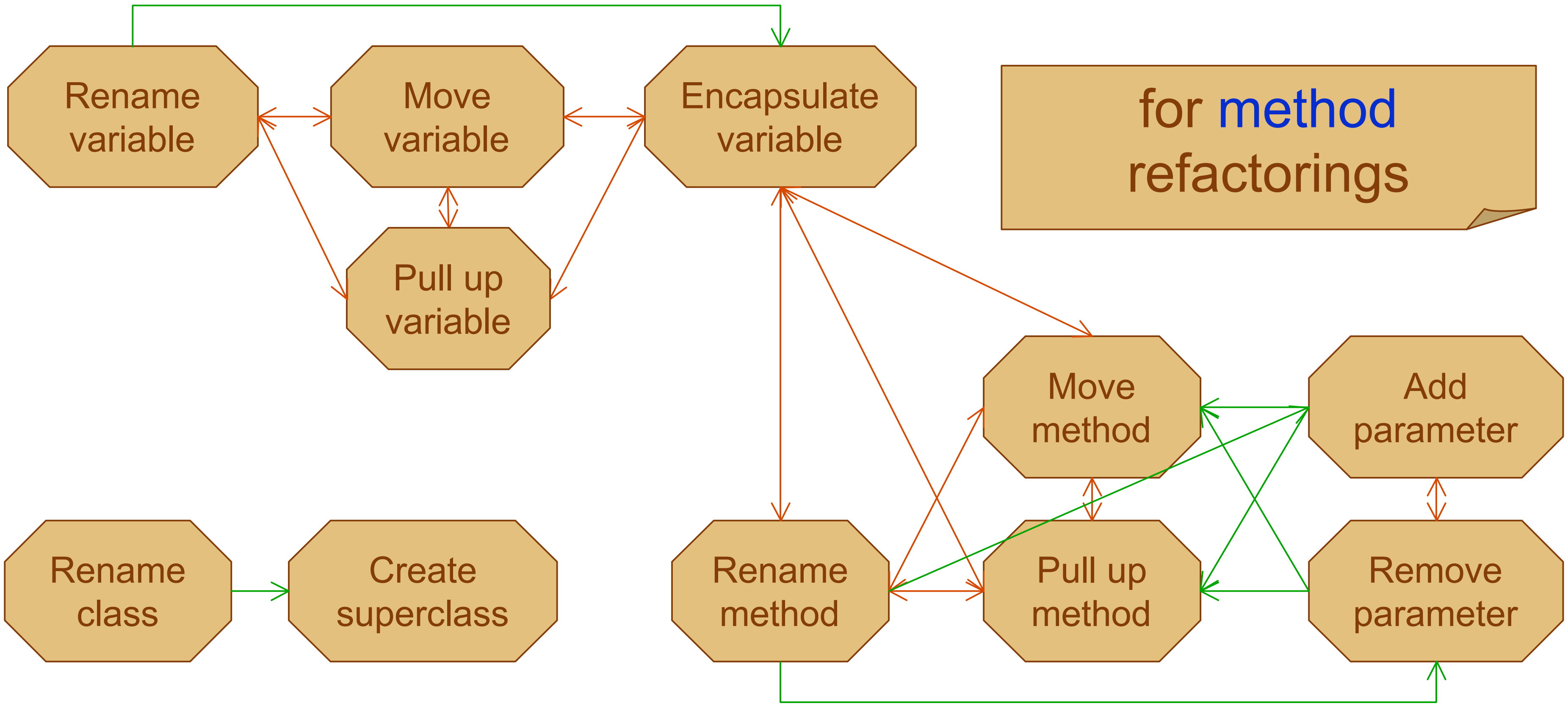
Refactoring dependencies

Step 4: Analyse dependency graph



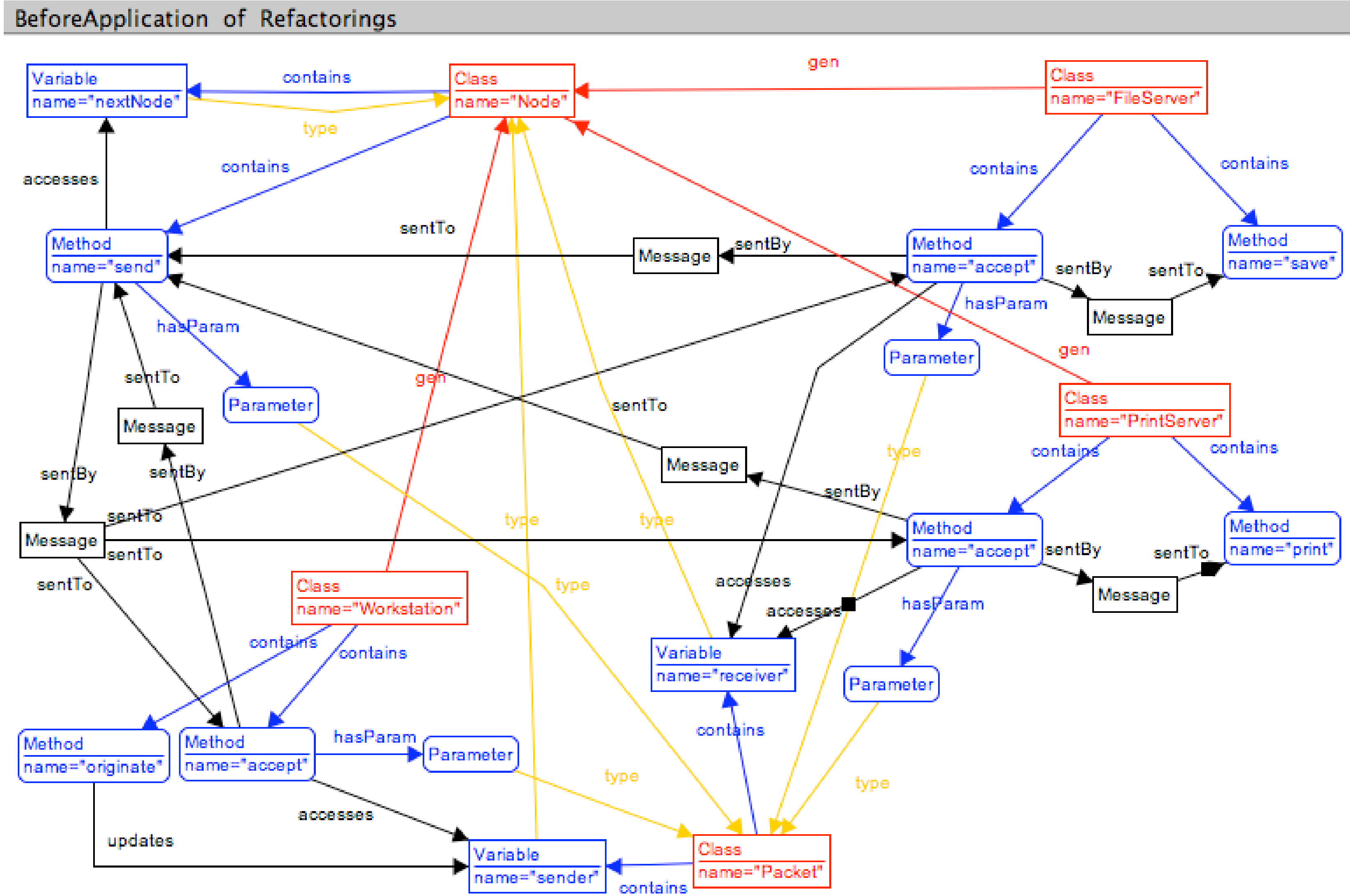
Refactoring dependencies

Step 4: Analyse dependency graph



Refactoring dependencies

Step 5: Fine-tune critical pairs in context of concrete input graph



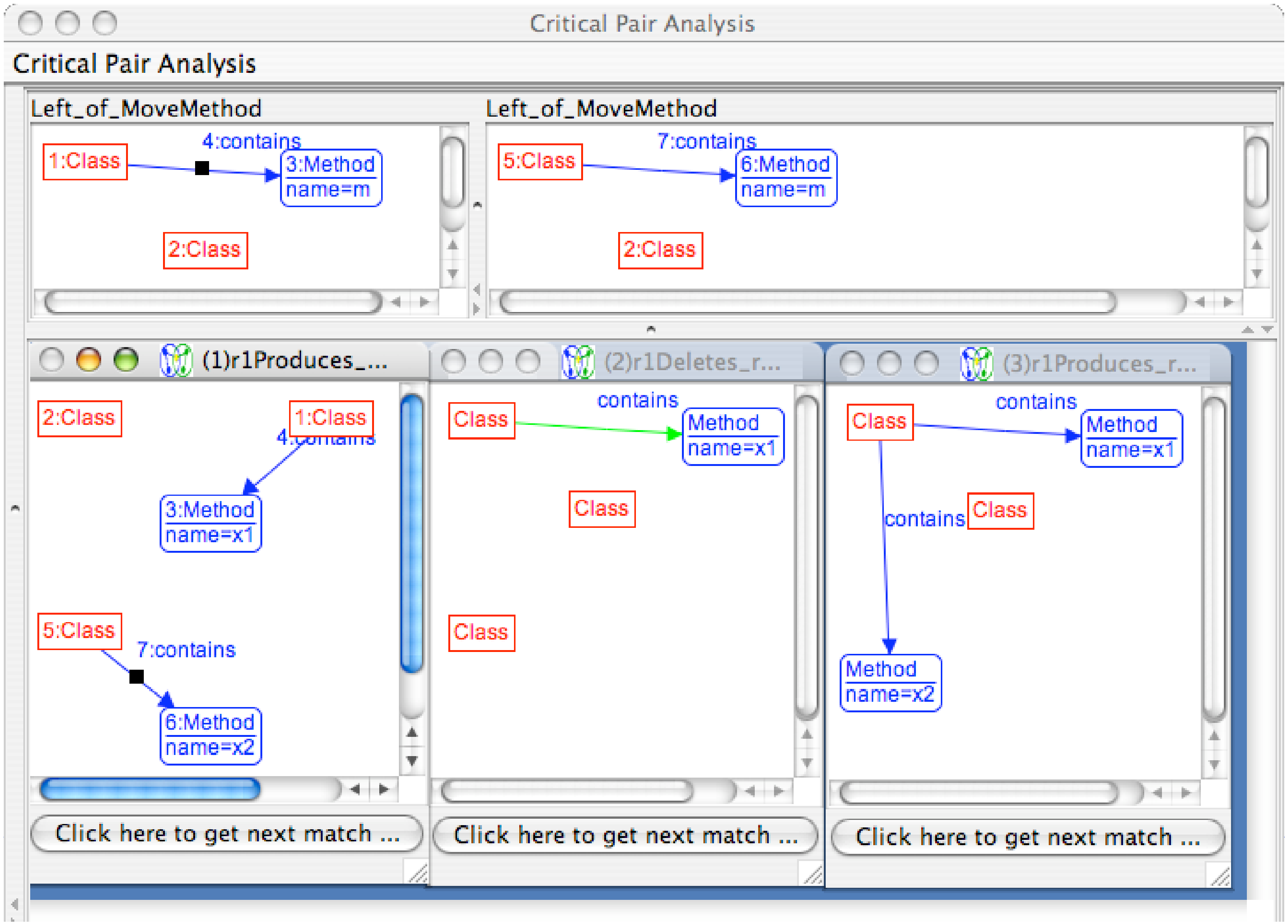
Refactoring dependencies

Step 5: Fine-tune critical pairs in context of concrete input graph

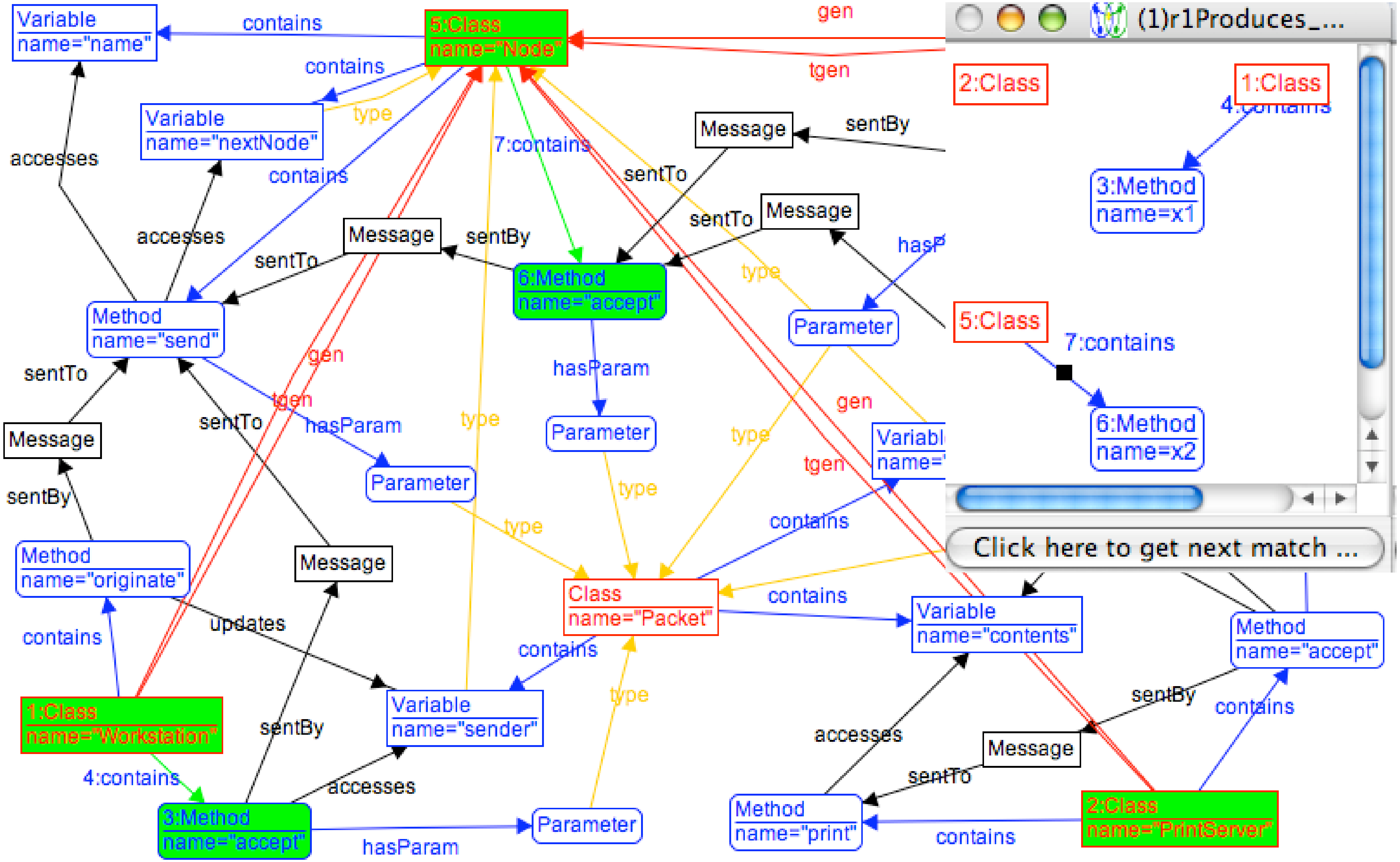
Critical Pairs

first \ second	1: MoveV...	2: MoveM...	3: PullUp...	4: PullUp...	5: CreateS...	6: Encaps...	7: AddPar...	8: Remov...	9: Renam...	10: Rena...	11: Rena...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	2	0	0	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	0	0	0	0	0	0	0	0
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	0	1	1	0	0	2

Refactoring dependencies



Refactoring dependencies

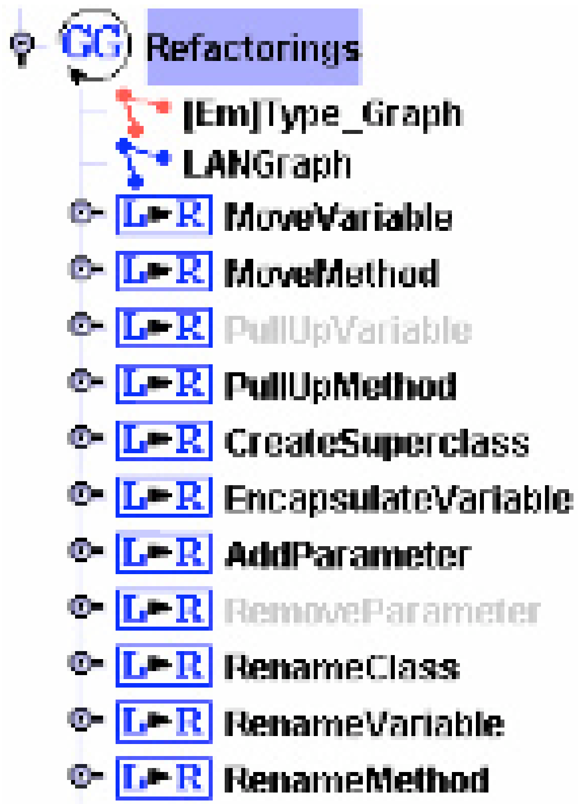


Refactoring dependencies

- Step 6: Perform sequential dependency analysis

To identify dependencies between refactorings that are applicable

Not fully supported in *AGG*



Generating Refactoring Code

1. Experiment in *Fujaba*

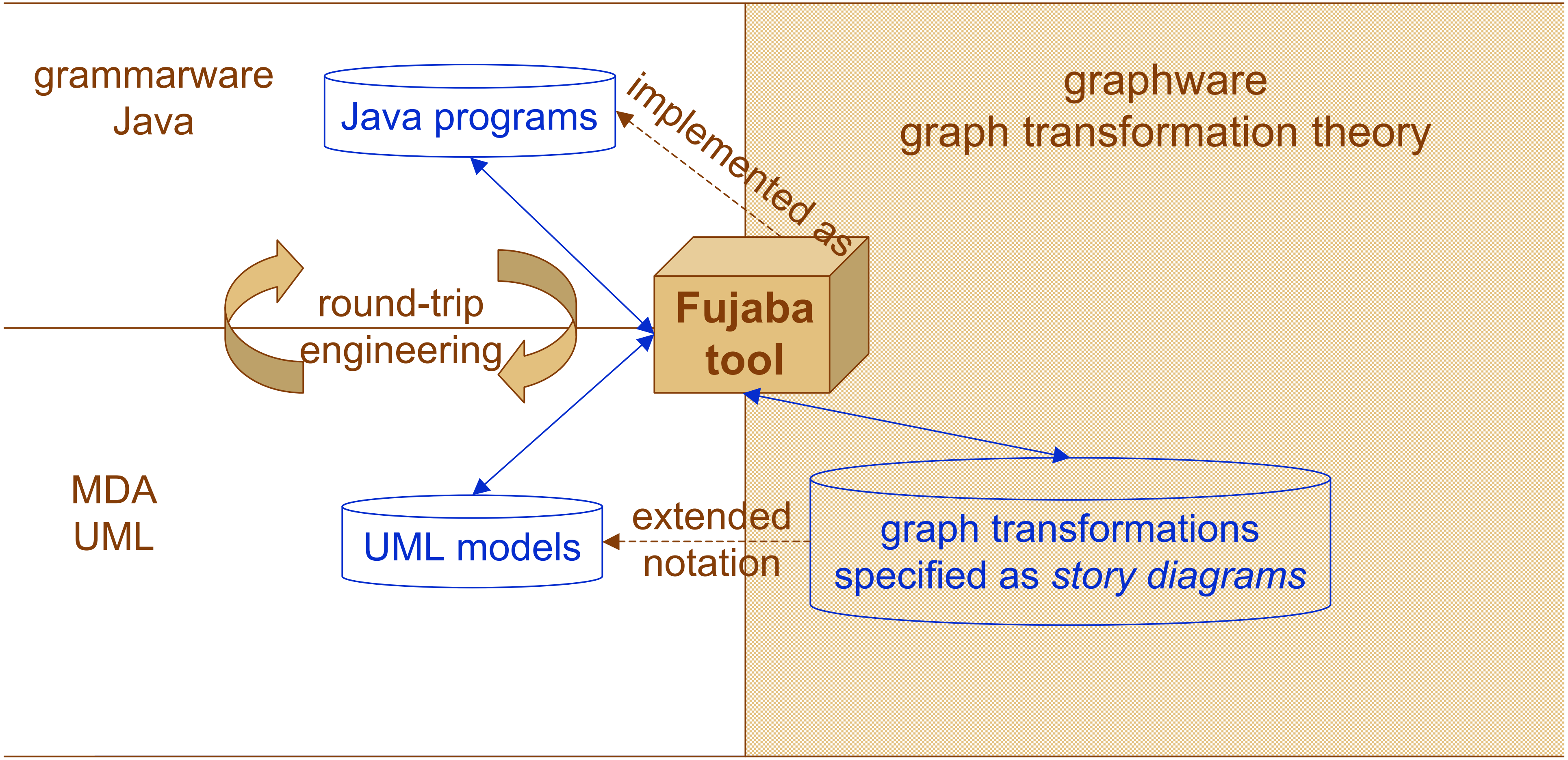
(in collaboration with *Pieter Van Gorp*
and *Niels Van Eetvelde*, University of Antwerp)

Generating refactoring code

- About *Fujaba* (From UML to Java and Back Again)
  - Round trip engineering with UML, Java, and design patterns
  - Class, collaboration and activity diagrams for story diagrams
    - Dynamic behavior
    - Automatic generation
  - Reverse engineering

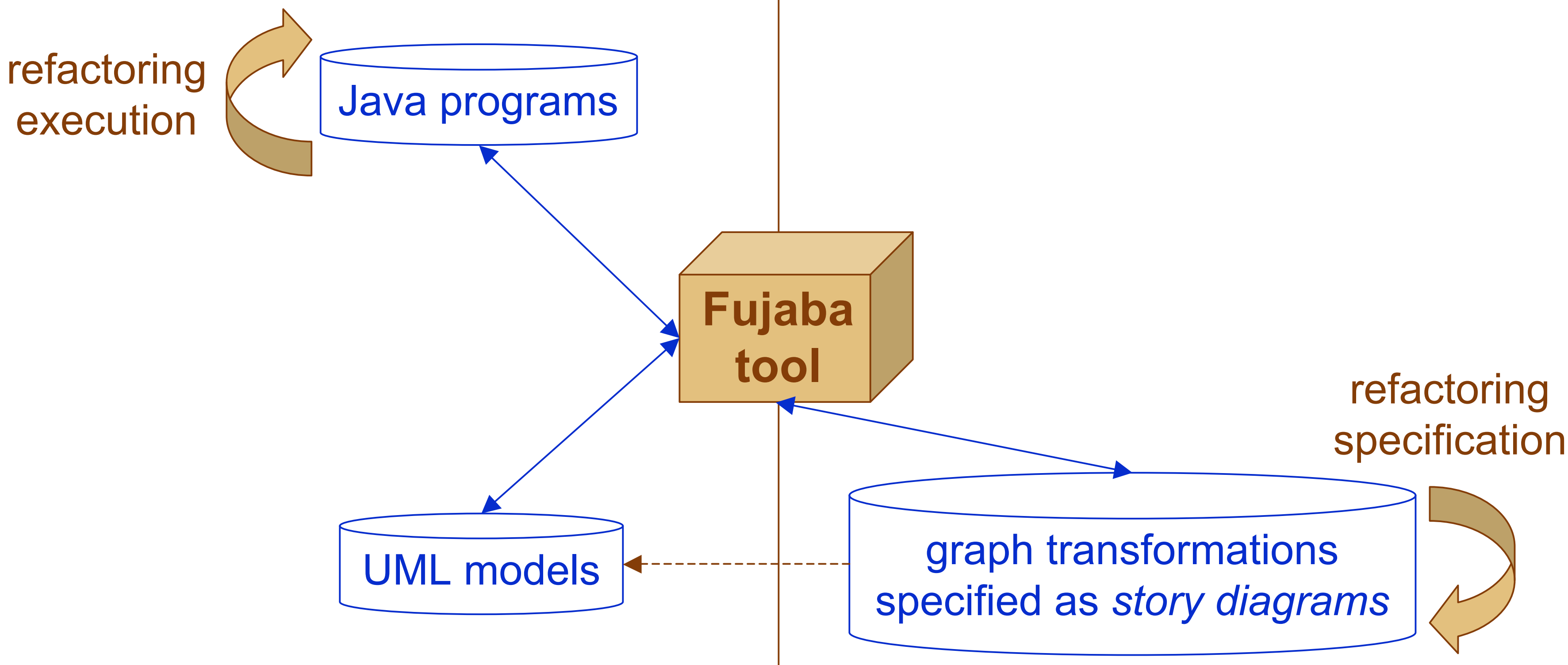
Generating refactoring code

- About *Fujaba* : Technological Spaces



Generating refactoring code

- About *Fujaba* : Technological Spaces



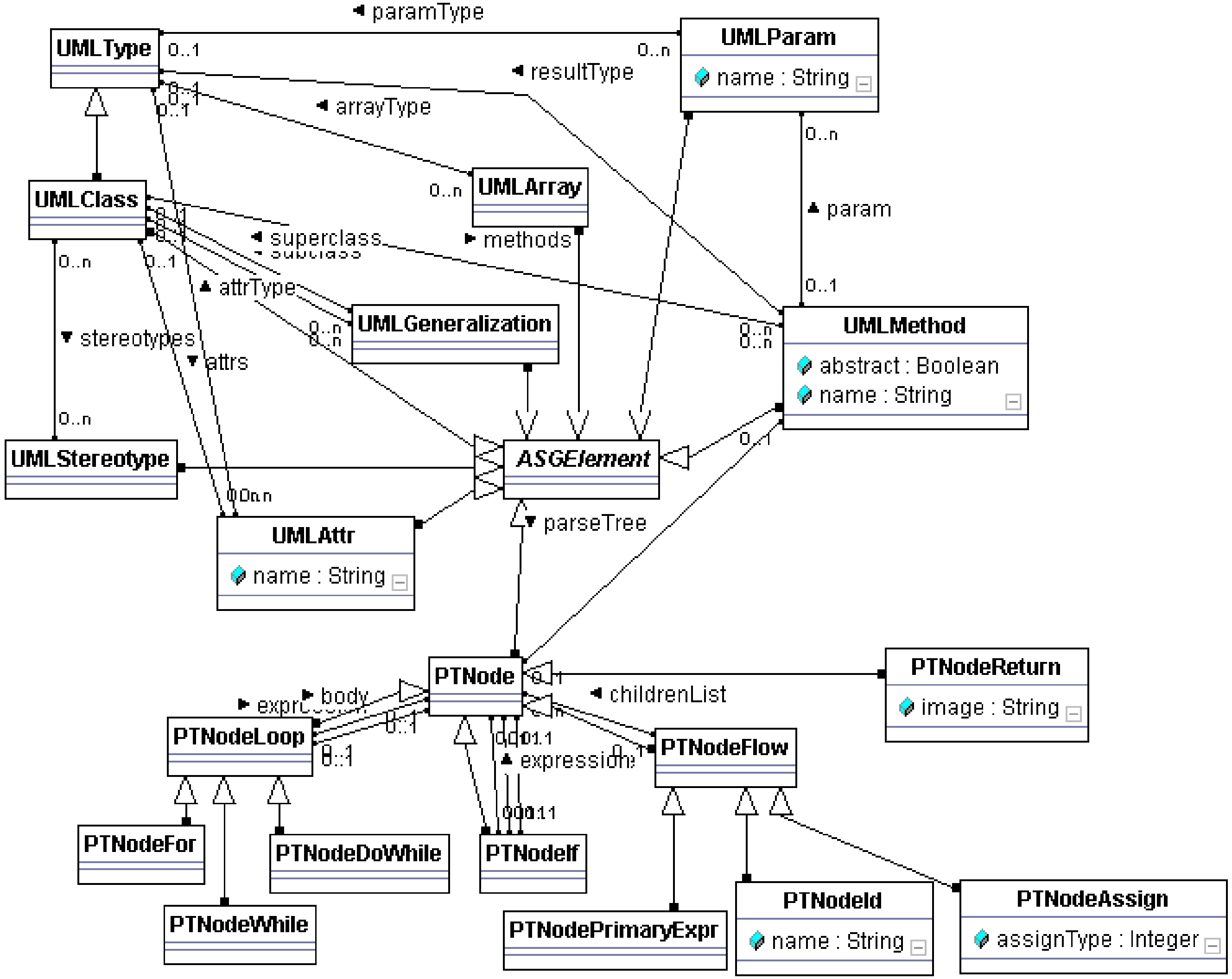
Generating Refactoring Code

- Experiment in *Fujaba*
  - Specify refactorings as *Fujaba* graph transformations using story diagram notation
  - Generate refactoring code from these transformations

- Advantages
  - easier to specify and understand refactorings (visual notation)
  - easier to implement refactorings (automatic code generation)

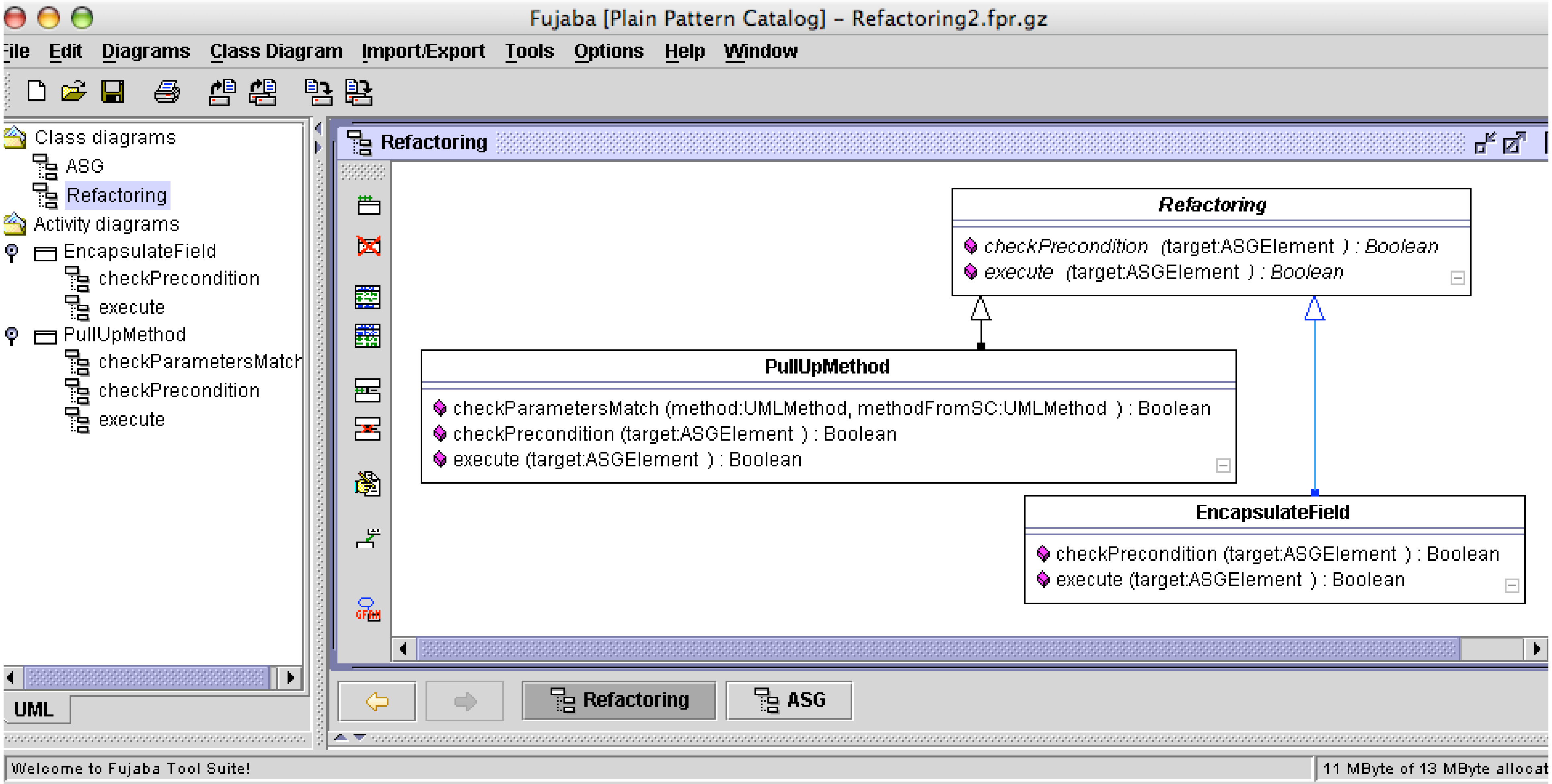
Generating Refactoring Code

- *Fujaba's metamodel*

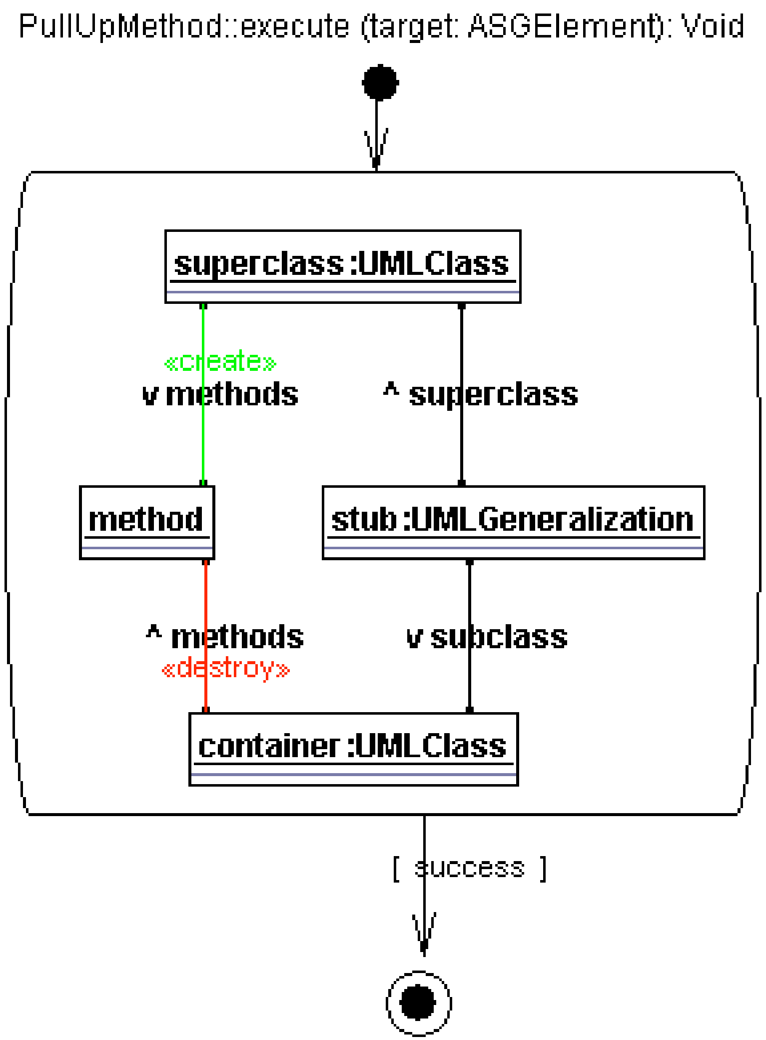


Generating Refactoring Code

- Refactoring framework in *Fujaba*



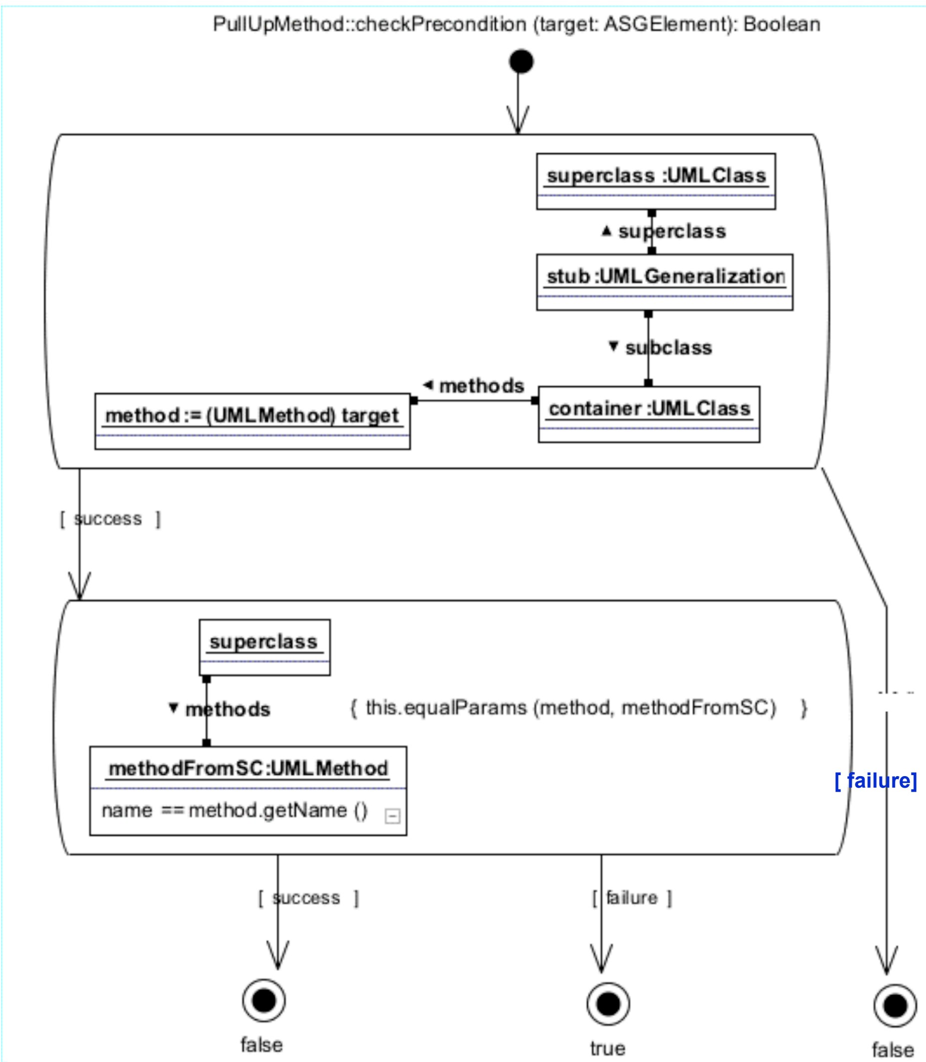
Generating Refactoring Code

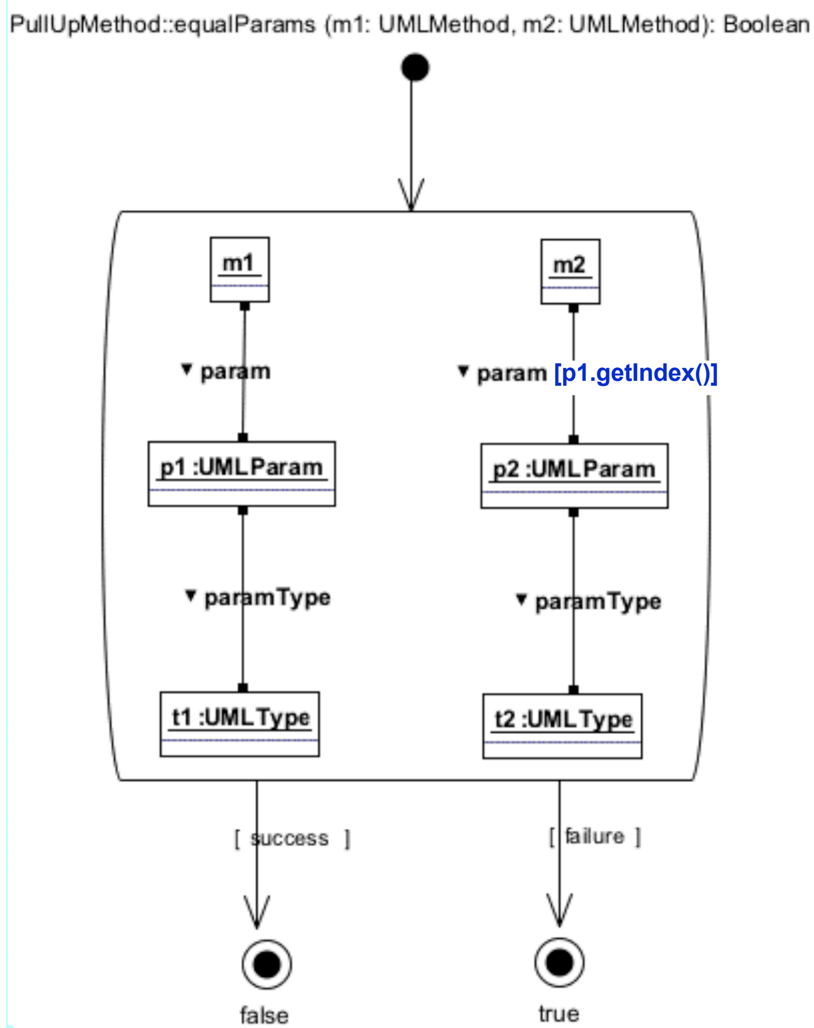


***Pull Up Method* model refactoring**
as Graph Transformation

1. Match `method`
   > (Hidden) Cast target
2. Match `container`
   > Link Navigation
3. Match `stub`
   > Link Navigation
4. Match `superclass`
   > Link Navigation
5. Remove `method` from container
6. Add `method` to superclass

Generating Refactoring Code

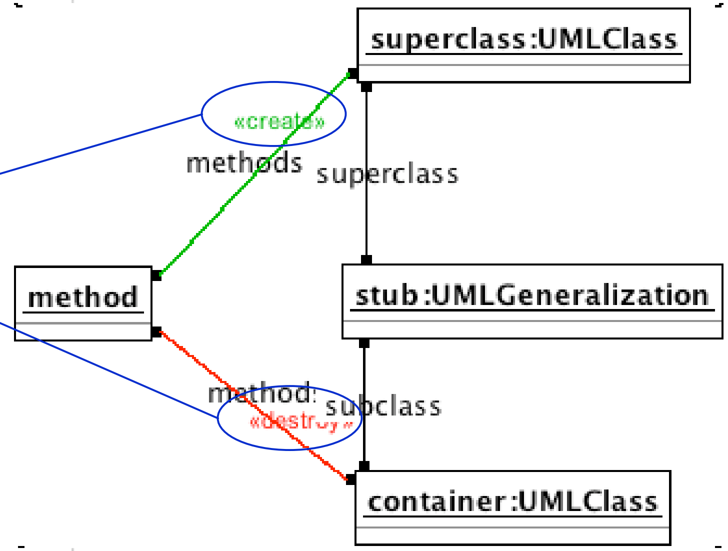




Generating Refactoring Code

Editor

```
        // bind stub : UMLGeneralization
        fujaba__IterContainerRevSubclassStub = container.iteratorOfRevSubclass () ;
        while ( !(fujaba__Success) && fujaba__IterContainerRevSubclassStub.hasNext () )
        {
           try
           {
              stub = (UMLGeneralization) fujaba__IterContainerRevSubclassStub.next () ;

              // bind superclass : UMLClass
              // sourceObjectName=stub,sourceObjectType=UMLGeneralization,sourceRoleName=superclass,targetObjectName=superclass,tar
              superclass = stub.getSuperclass () ;
              JavaSDM.ensure ( superclass != null ) ;

              // check isomorphic binding
              JavaSDM.ensure ( !(container.equals (superclass)) ) ;

              // delete link
              container.removeFromMethods (method);

              // create link
              superclass.addToMethods (method);

              fujaba__Success = true ;
           }
           catch ( JavaSDMException fujaba__InternalException )
           {
           }
        }
     }
     catch ( JavaSDMException fujaba__InternalException )
     {
        fujaba__Success = false ;
```



Generating Refactoring Code

Fujaba Plugin

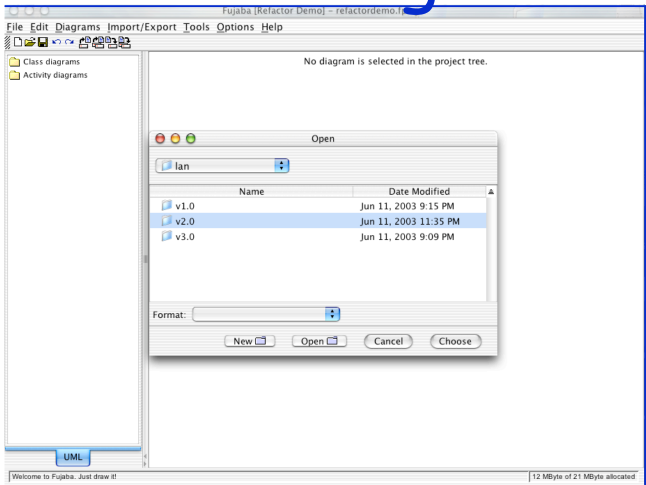
Editor

```java
/**...............................
public class PUMAction extends AbstractAction
{
   /**..................................
   public void actionPerformed (ActionEvent e
   {
      Object source = event.getSource();
      if (source instanceof Iterator)
      {
         Iterator iter = (Iterator) source;
         if (iter.hasNext())
         {
            source = iter.next();
         }
      }

      if (source instanceof UMLMethod)
      {
         UMLMethod m =  (UMLMethod) source;
         System.out.println("Pulling up " + m.
         // TODO: make it all static or pass ModelElement to constructor of Refactoring
         PullUpMethod pum= new PullUpMethod();
         if (pum.checkPrecondition(m)) {
            pum.execute(m);
            UMLProject.get().refreshDisplay();
            FrameMain.get().createNewTreeItems();
         } else {
               JOptionPane.showMessageDialog(null, // not dependent on parent frame
                    "Unable to execute refactoring: Preconditions not met.");
         }
      }
   } // executeAction
```

```xml
<Action id="doPUM" class="be.ac.ua.grammyuml.actions.PUMAction">
   <Name>Pull Up Method</Name>
   <ToolTip>Pull Up Method</ToolTip>
   <Icon>de/uni_paderborn/fujaba/app/images/none.gif</Icon>
</Action>

<PopupMenu class="de.uni_paderborn.fujaba.uml.UMLMethod">
  <MenuSection id="RefactorMenuSection">
    <Menu id="PUMMenu" actionId="doRefactorSTUB">
      <Name>Refactor</Name>
      <Icon>de/uni_paderborn/fujaba/app/images/none.gif</Icon>
      <MenuSection id="PUMMenuSection">
        <MenuItem actionId="doPUM"/>
      </MenuSection>
    </Menu>
  </MenuSection>
</PopupMenu>
```

Generating Refactoring Code

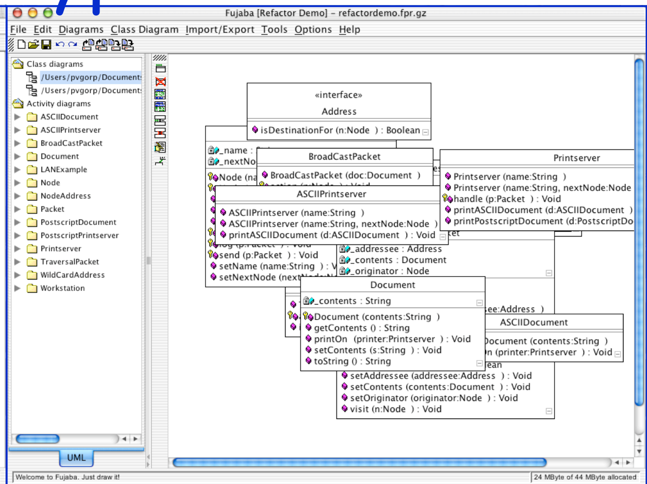
- Running Prototype



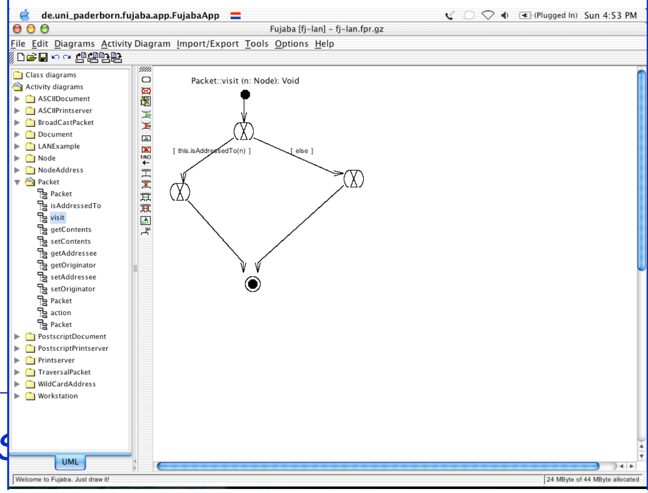
Import Java Sources (by directory)

1 parse

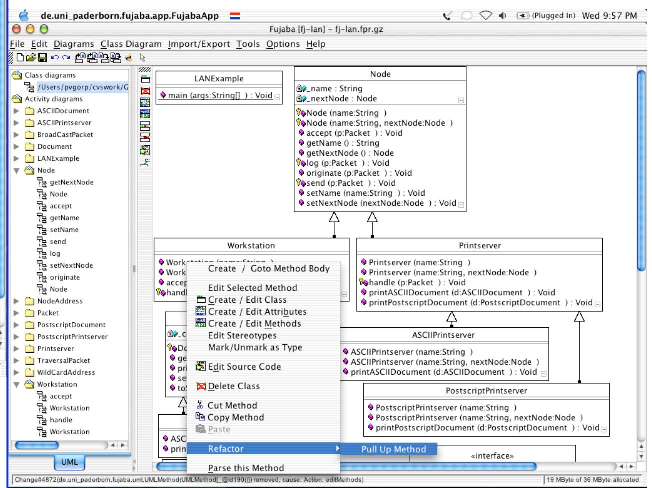
Old Sources



2 Rearrange diagrams



3 Execute Refactoring



4 regenerate

New Sources

Generating Refactoring Code

2. CASE-tool independent approach

(by *Pieter Van Gorp* and *Hans Schippers*, University of Antwerp)

Generating refactoring code

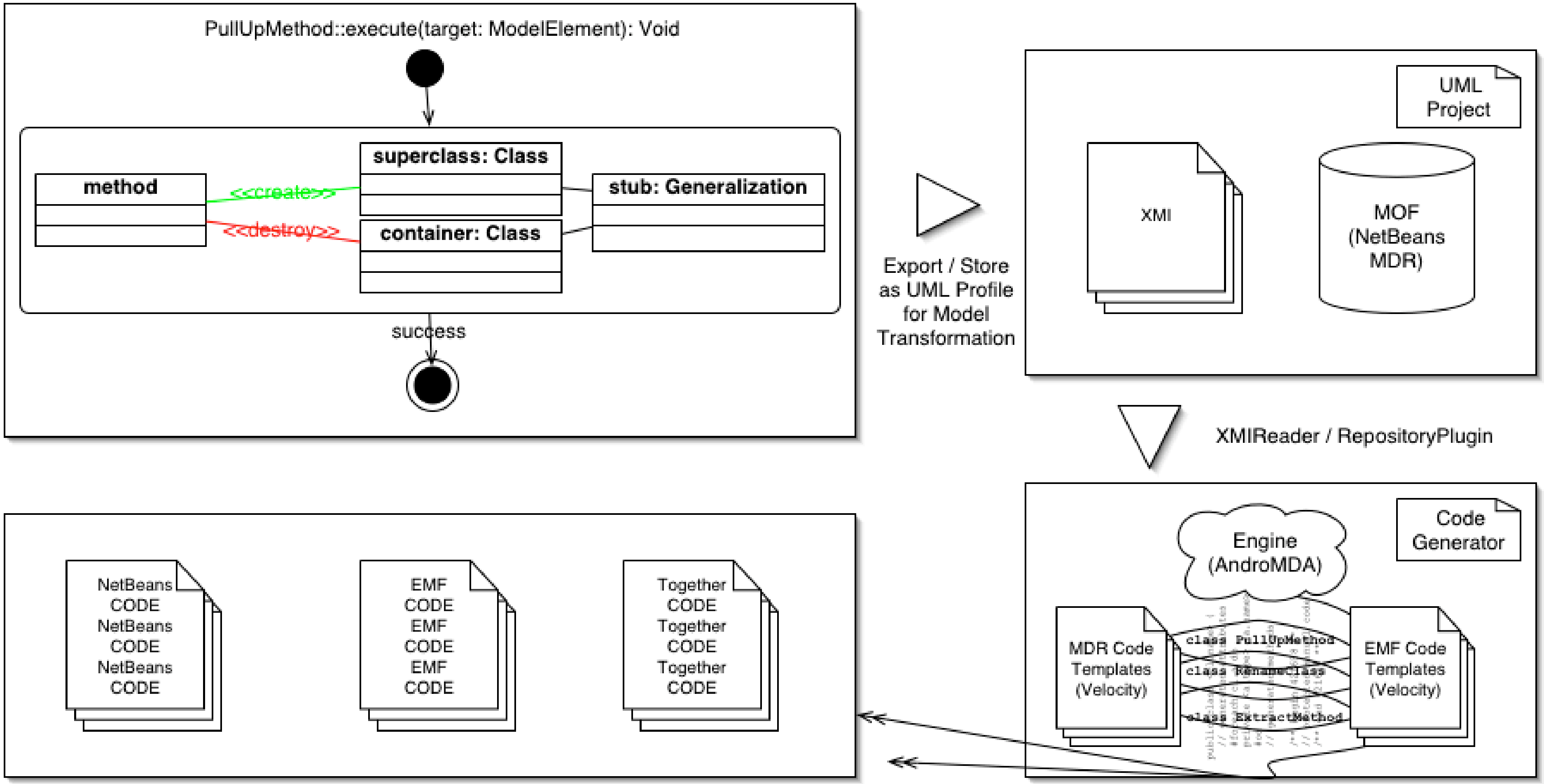
- Evaluation
  - *Fujaba* experiment was successful
    - Intuitive story diagram notation
    - GT can be used to express refactorings
    - Refactoring code can be generated
    - Refactoring plug-ins can be written
  - But…
    - *Fujaba's* internal metamodel not MOF/UML compliant
      - Generated code not reusable in other MDE tools
    - Story diagram notation only available in *Fujaba*
      - No commercial, industrial support

Generating refactoring code

- Solution
  - Provide UML profile support for story diagram notation
  - Make generated refactoring code CASE tool independent
    - using MOF, XMI, JMI, EMF, …

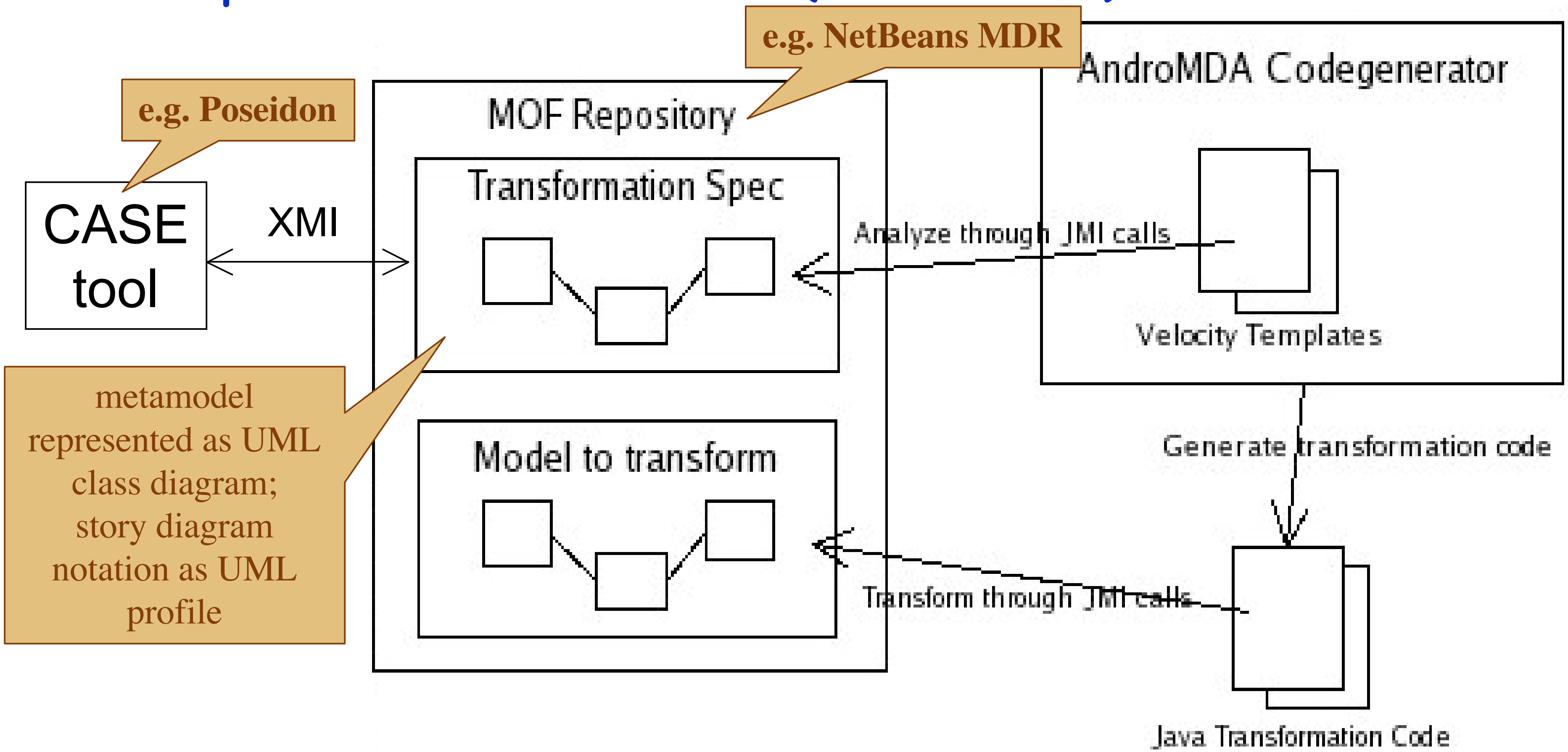
Generating refactoring code

- Proposed architecture

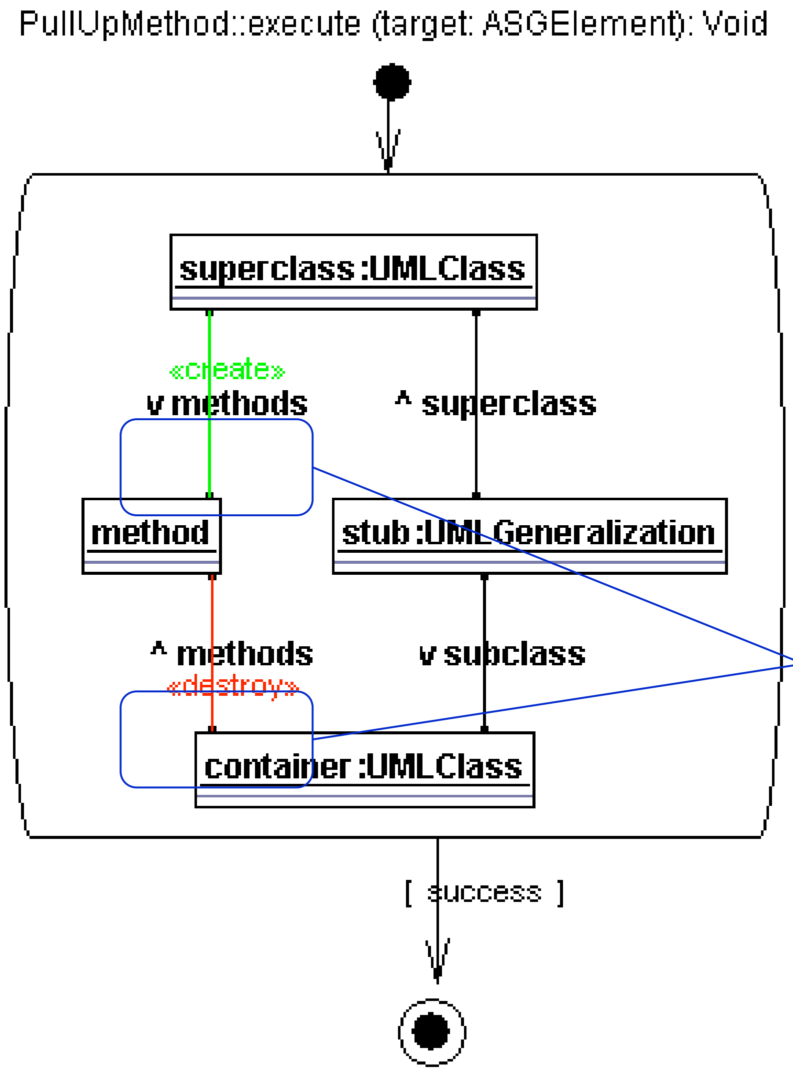


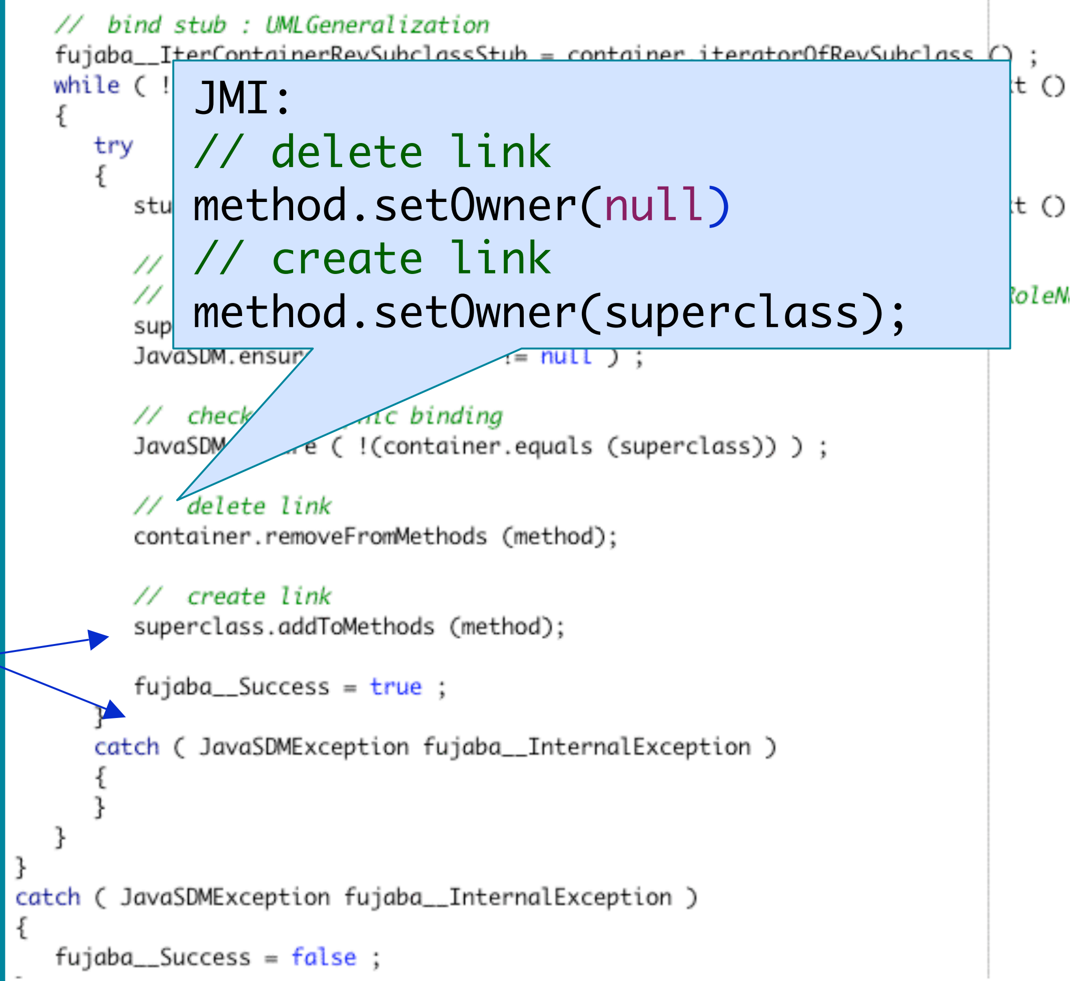
Generating refactoring code

- Proposed architecture (continued)

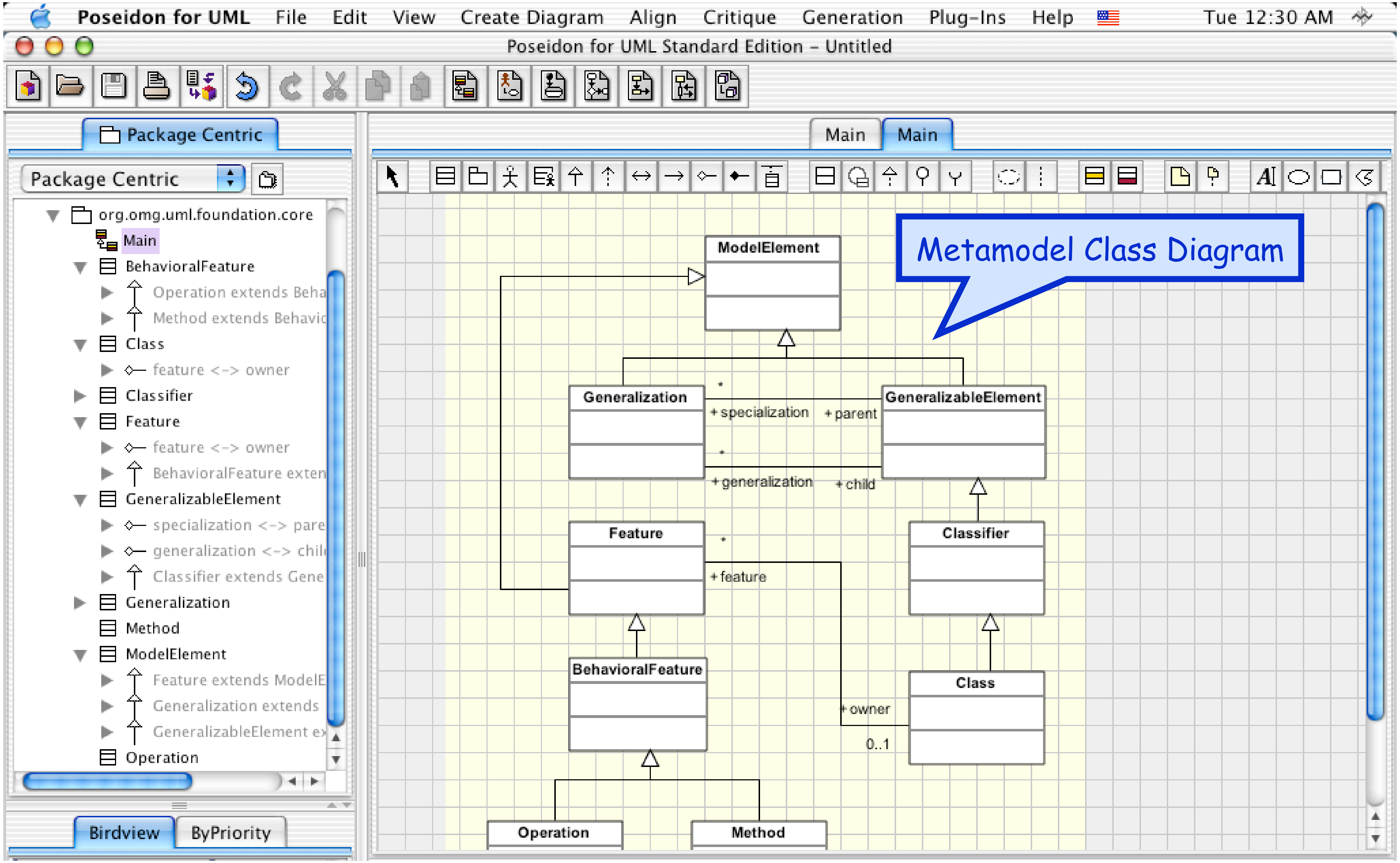


Generating refactoring code
Fujaba versus JMI code

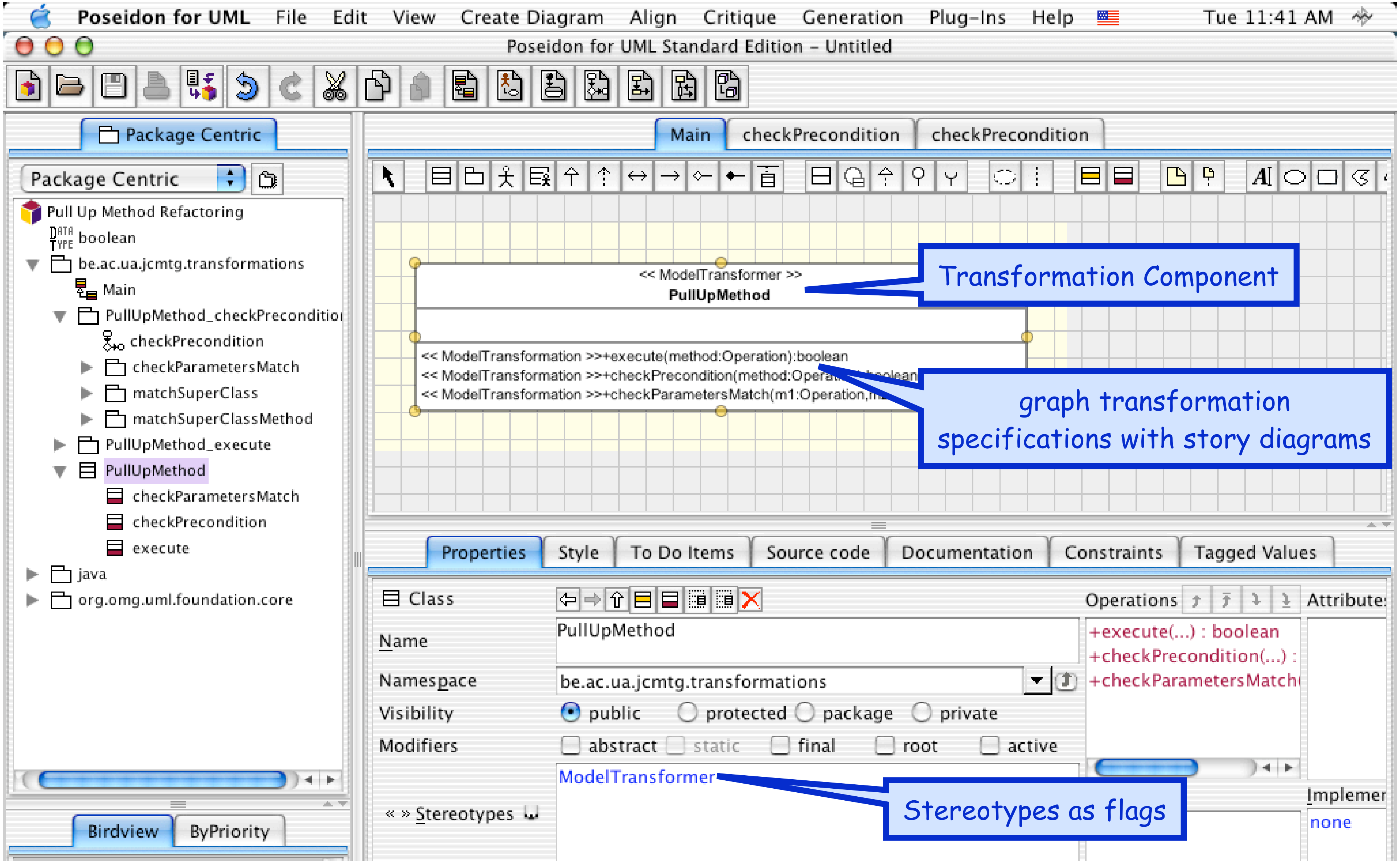




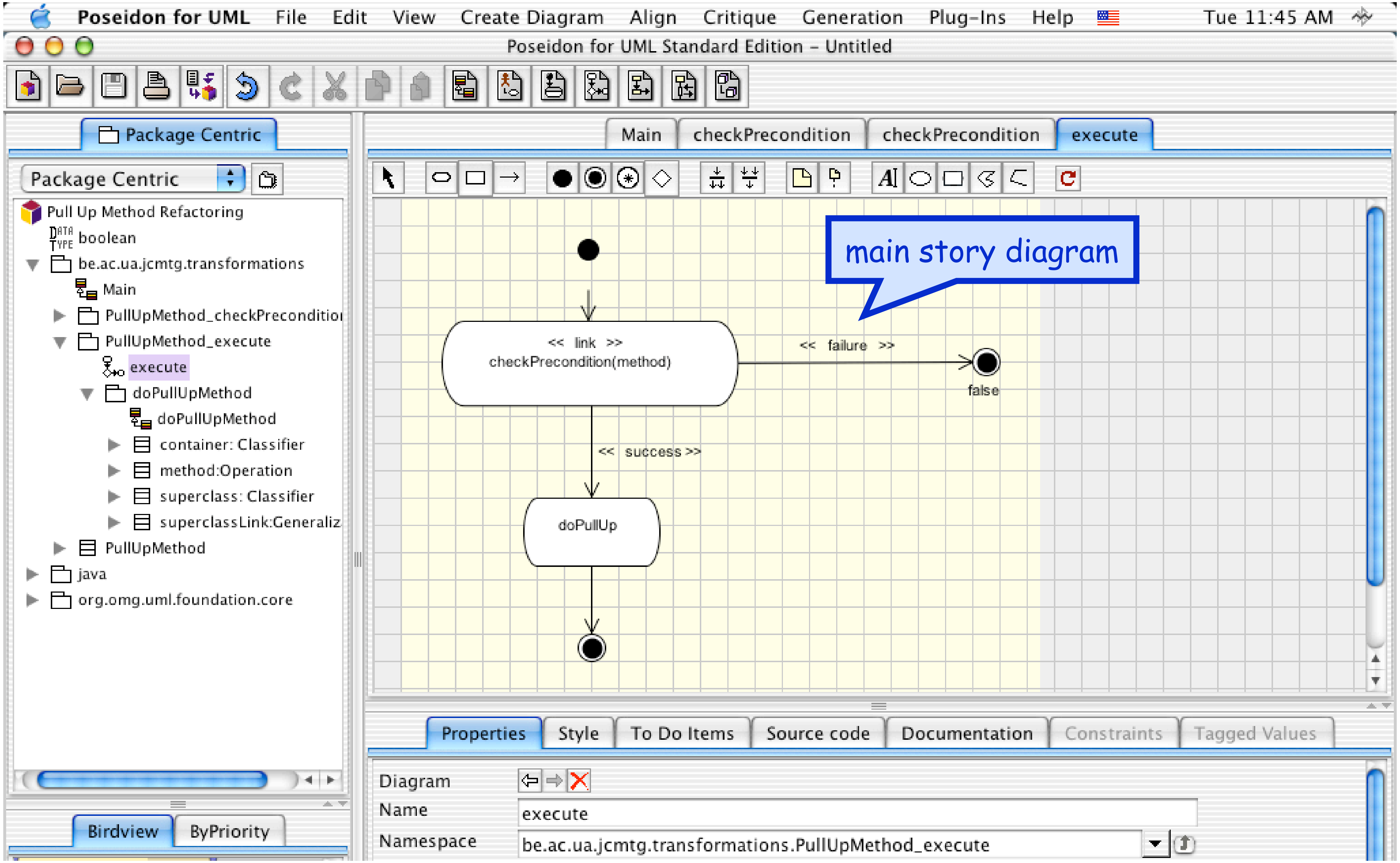
Generating refactoring code
Poseidon "proof of concept"



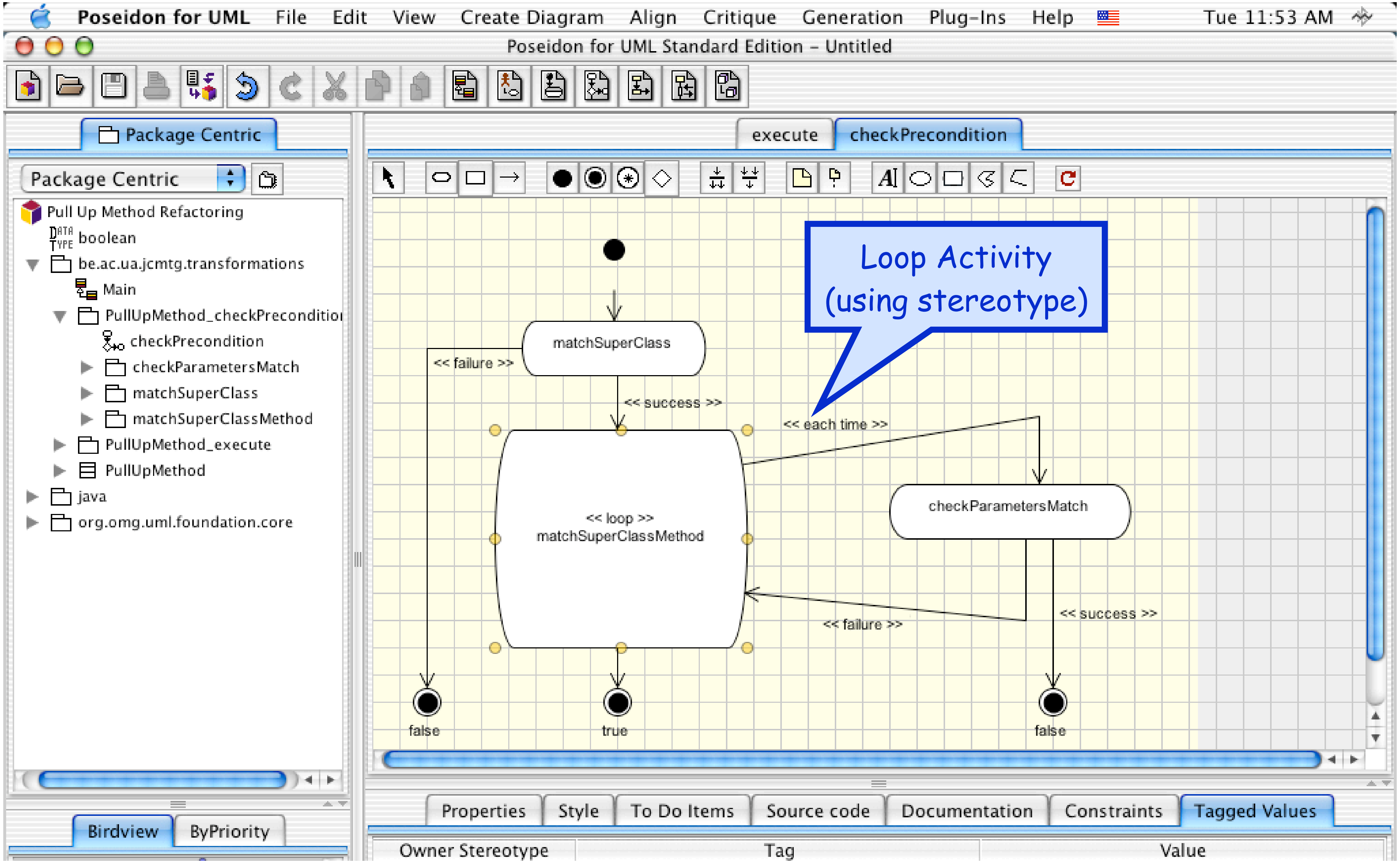
Generating refactoring code
Poseidon "proof of concept"



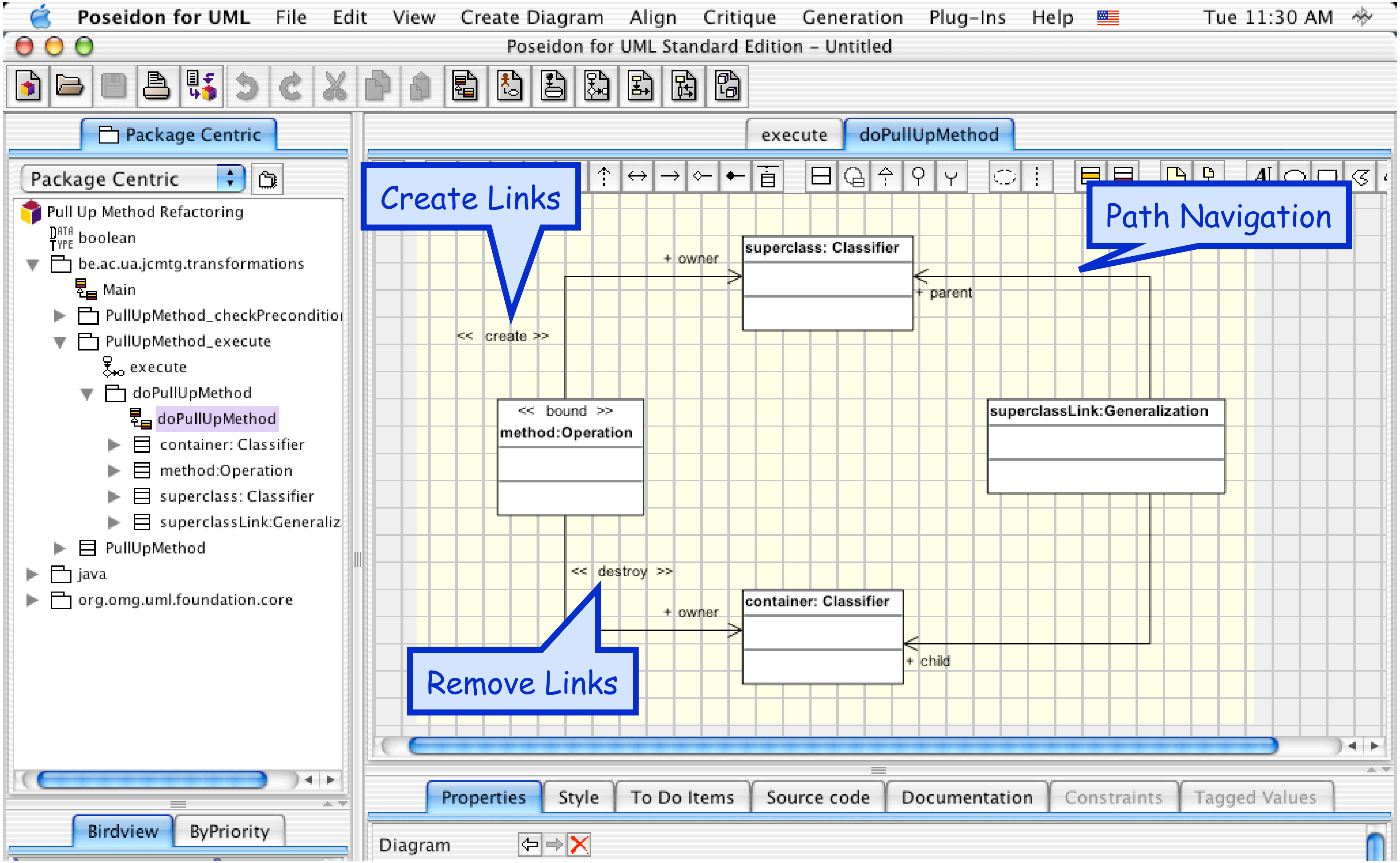
Generating refactoring code
Poseidon "proof of concept"



Generating refactoring code
Poseidon “proof of concept”

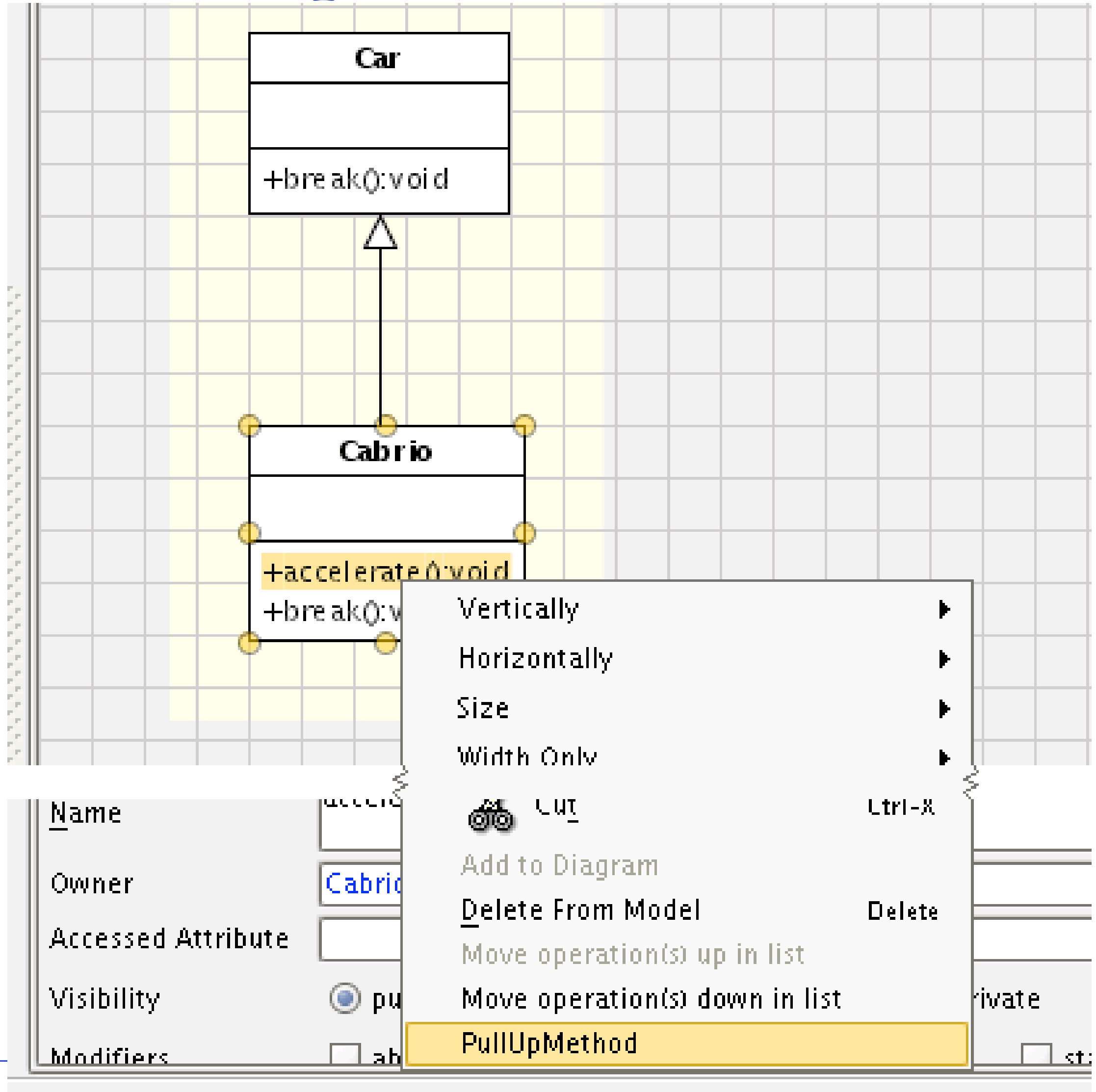


Generating refactoring code
Poseidon "proof of concept"



Generating refactoring code Poseidon "proof of concept"

- Poseidon Plugin
  - Extra menu item for launching refactoring transformation
  - generated from transformation model



Generating refactoring code

- CASE tool independent approach is feasible
  - Using the proposed architecture
    - UML profiles for story diagram notation
  - Illustrated through Poseidon "proof of concept"
  - Can be repeated for other case tools
    - Magicdraw, Together, Objecteering, Poseidon, …
  - Can be compared to, or integrated with, other MDE frameworks
    - EMF support, ATL framework, …

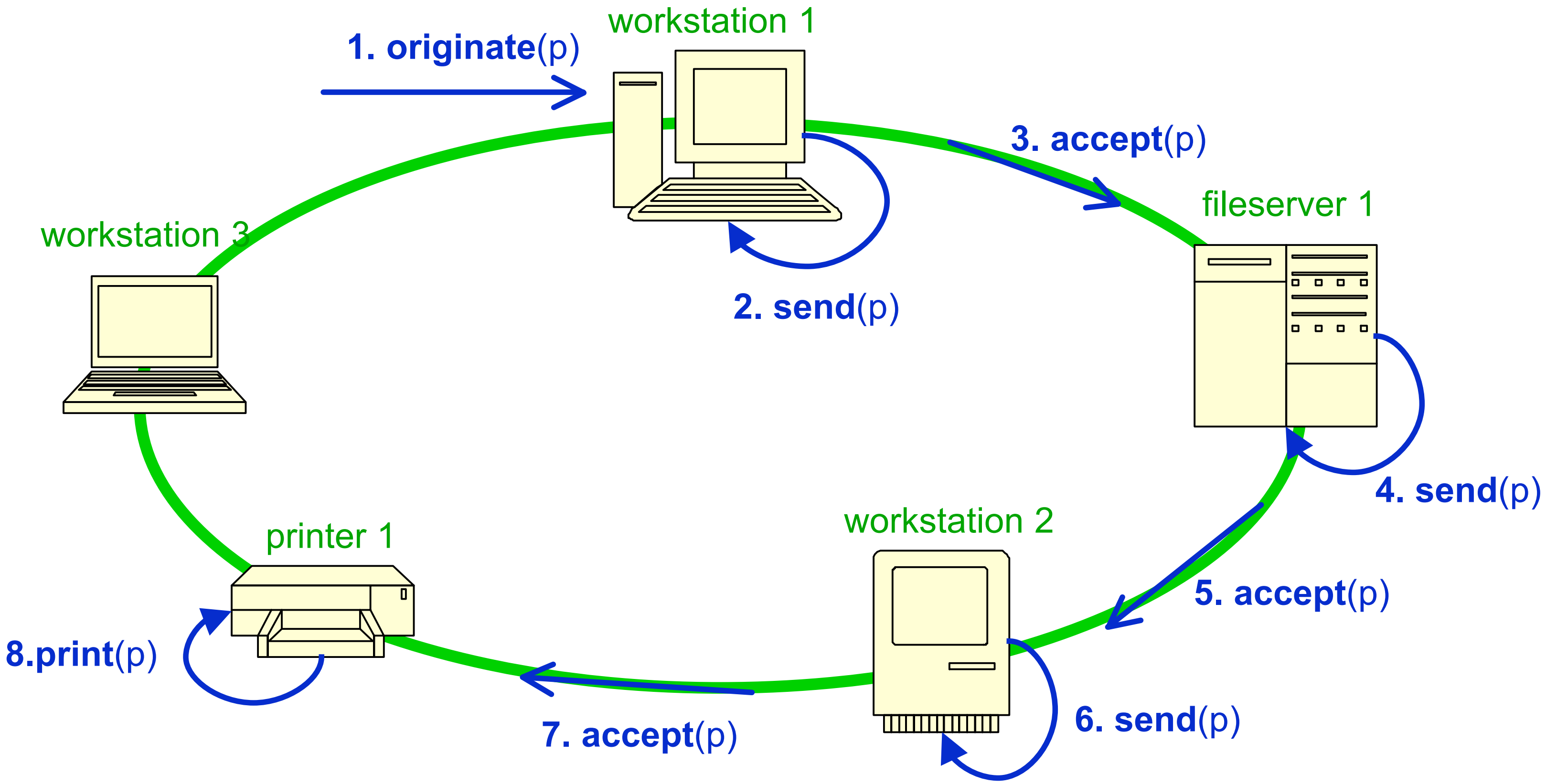
Behaviour preservation of refactorings

formal experiment

(in collaboration with Dirk Janssens
and Serge Demeyer, University of Antwerp)

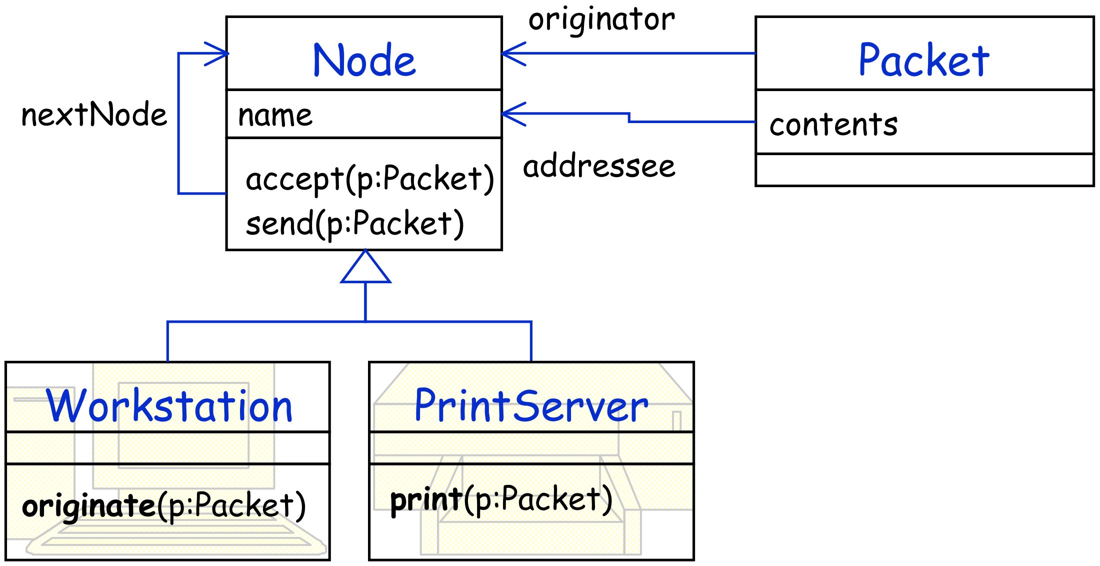
Formal experiment
Behaviour preservation

- Case study: LAN simulation



Formal experiment
Behaviour preservation

- UML class diagram



Formal experiment
Behaviour preservation

- Java source code

```
public class Node {
  public String name;
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      name +
      "sends to" +
      nextNode.name);
    nextNode.accept(p); }
  }
```

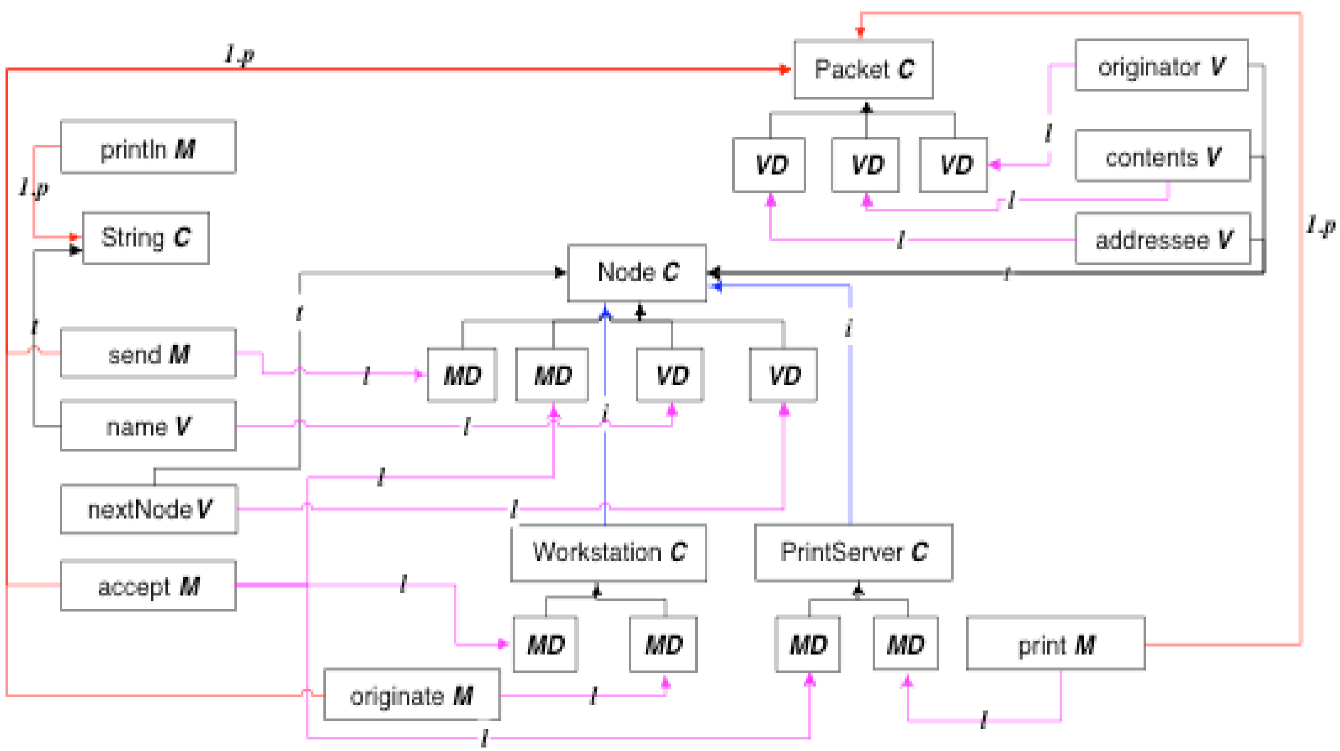
```
public class Packet {
  public String contents;
  public Node originator;
  public Node addressee;
  }
```

```
public class Printserver extends Node {
  public void print(Packet p) {
    System.out.println(p.contents);
    }
  public void accept(Packet p) {
    if(p.addressee == this)
      this.print(p);
    else
      super.accept(p);
    }
  }
```

```
public class Workstation extends Node {
  public void originate(Packet p) {
    p.originator = this;
    this.send(p);
    }
  public void accept(Packet p) {
    if(p.originator == this)
      System.err.println("no
destination");
    else super.accept(p);
    }
  }
```

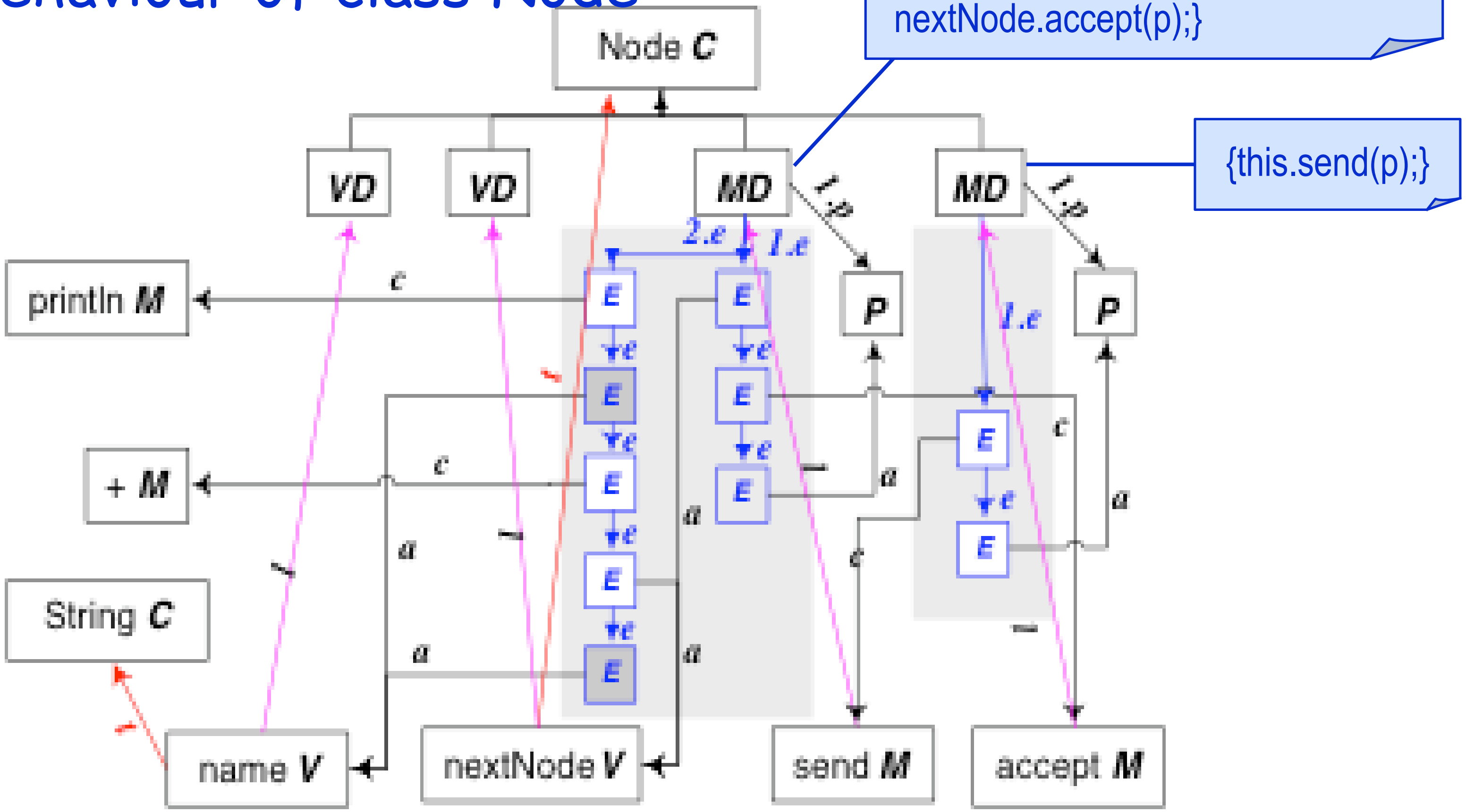
Formal experiment
Behaviour preservation

- Program structure
  - Directed, labelled, typed graph



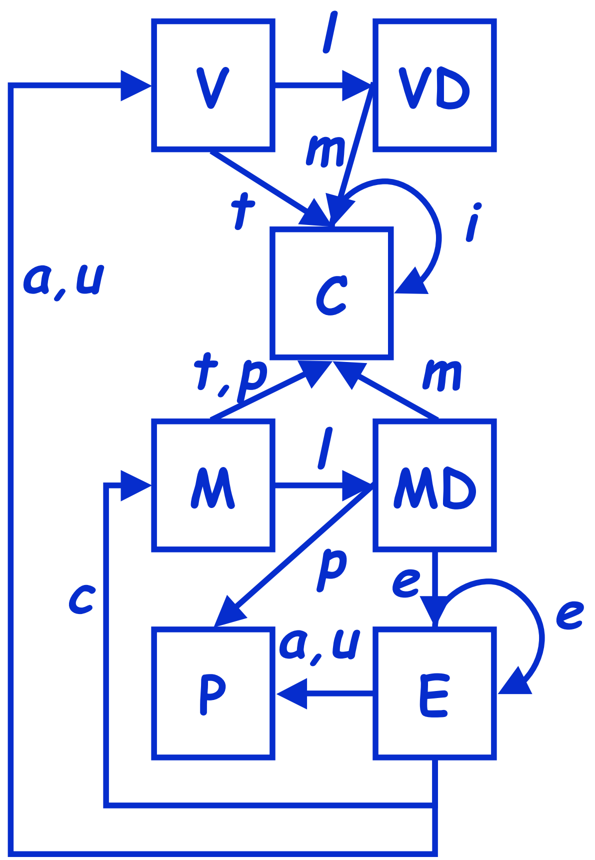
Formal experiment
Behaviour preservation

- Program behaviour
  - Behaviour of class Node



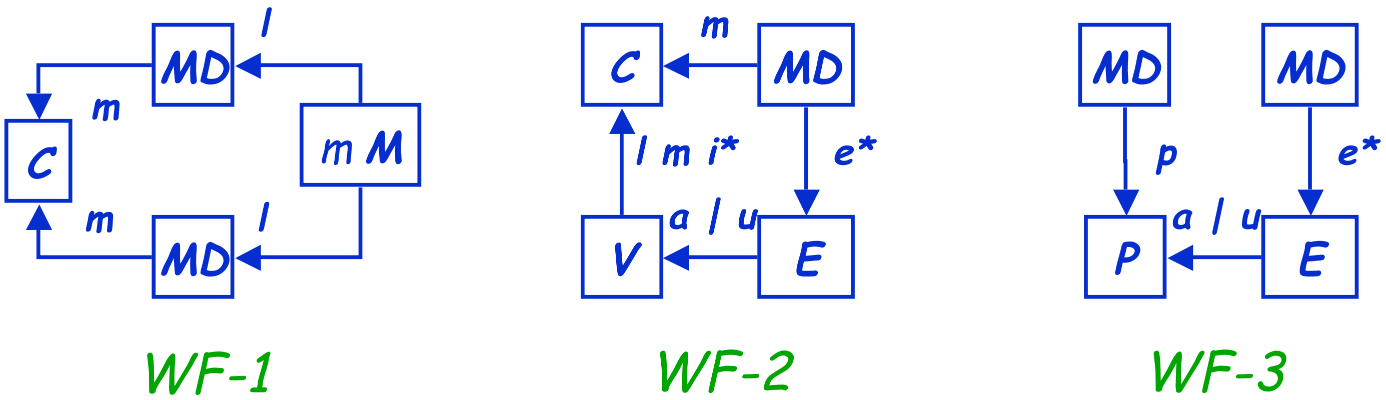
Formal experiment
Behaviour preservation

- *Type graph*
  - *Represents OO metamodel*
  - *Expresses well-formedness constraints on the graph model*



Formal experiment
Behaviour preservation

- Forbidden subgraphs
  - *WF-1*: a class cannot define the same method twice
  - *WF-2*: a method cannot refer to variables in descendant classes
  - *WF-3*: a method cannot refer to parameters of other methods



Formal experiment
Behaviour preservation

- *Pull up method*
  - Replace similar methods in subclasses by common superclass method
  - Precondition
    - Replaced method should not refer to methods in subclasses

Formal experiment
Behaviour preservation

- *EncapsulateVariable* encapsulates public variables and provides accessor methods

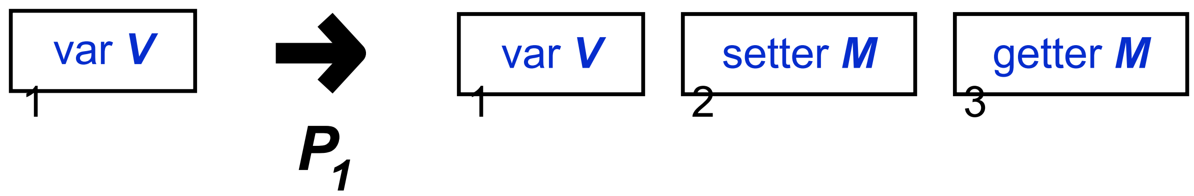
```
public class Node {
  public String name;
  public Node nextNode;
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      name +
      "sends to" +
      nextNode.name);
    nextNode.accept(p); }
  }
```

```
public class Node {
  private String name;
  private Node nextNode;
  public String getName() {
    return this.name; }
  public void setName(String s) {
    this.name = s; }
  public Node getNextNode() {
    return this.nextNode; }
  public void setNextNode(Node n) {
    this.nextNode = n; }
  public void accept(Packet p) {
    this.send(p); }
  protected void send(Packet p) {
    System.out.println(
      this.getName() +
      "sends to" +
      this.getNextNode().getName());
    this.getNextNode().accept(p); }
  }
```

Formal experiment
Behaviour preservation

- EncapsulateVariable(var,getter,setter)
  - *Parameterised* transformation
  - *Embedding* mechanism

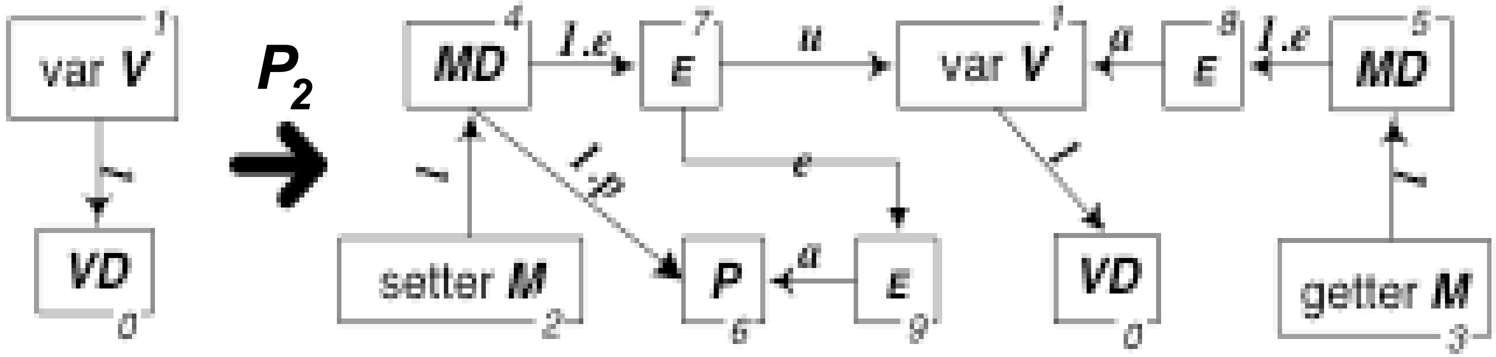
incoming edges	outgoing edges
	(t,1) → (t,1), (p,2), (t,3)



Formal experiment
Behaviour preservation

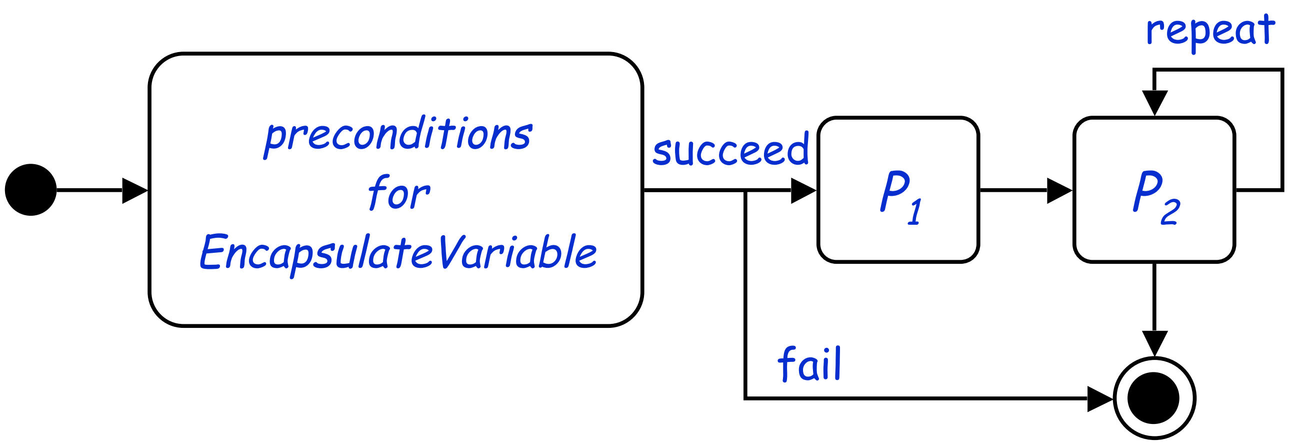
- EncapsulateVariable(var,getter,setter)
  - *Parameterised* transformation
  - *Embedding* mechanism

incoming edges	outgoing edges
(a,1) → (c,3) (u,1) → (c,2)	*(m,0) → (m,0), (m,4), (m,5)*



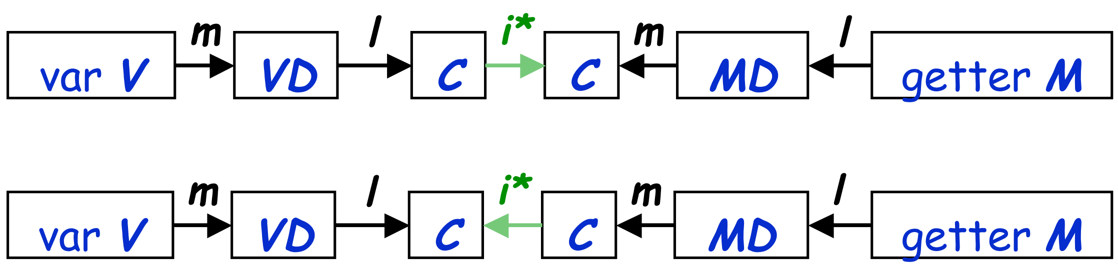
Formal experiment
Behaviour preservation

- *Controlled graph rewriting* is needed to
  - Control the application order of productions
  - Specify refactoring preconditions



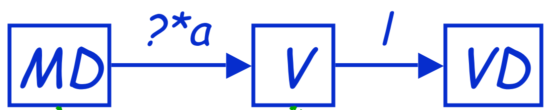
Formal experiment
Behaviour preservation

- Use preconditions
  - To satisfy wf-constraints
  - to satisfy more specific constraints
    - e.g. *EncapsulateVariable* should not introduce accessor method names that exist in inheritance chain
- Express *negative preconditions as forbidden subgraphs*

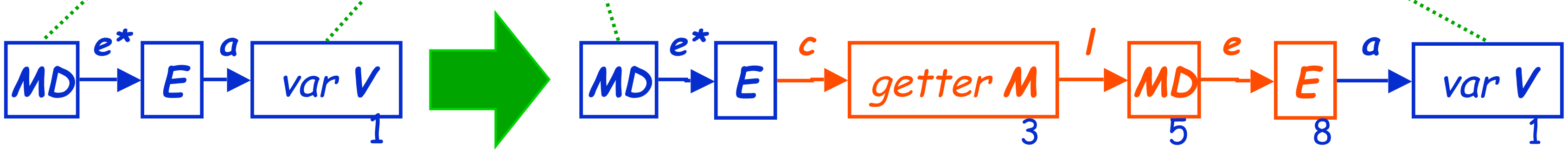


Formal experiment
Behaviour preservation

- Graph invariant

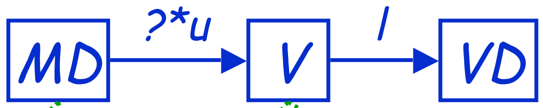


- *EncapsulateVariable* preserves access behaviour
  - variables remain accessible via transitive closure

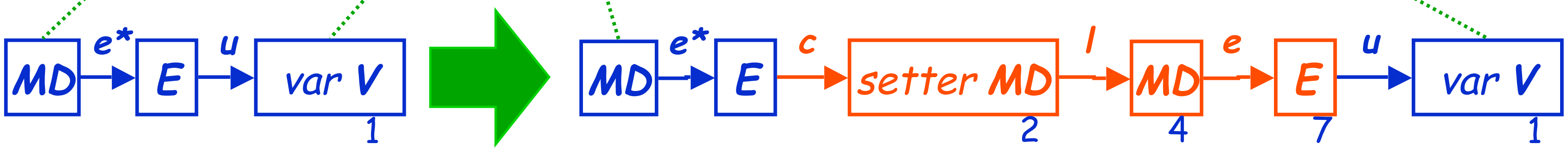


Formal experiment
Behaviour preservation

- Graph invariant



- *EncapsulateVariable* preserves update behaviour
  - variables remain updatable via transitive closure



Formal experiment
Behaviour preservation

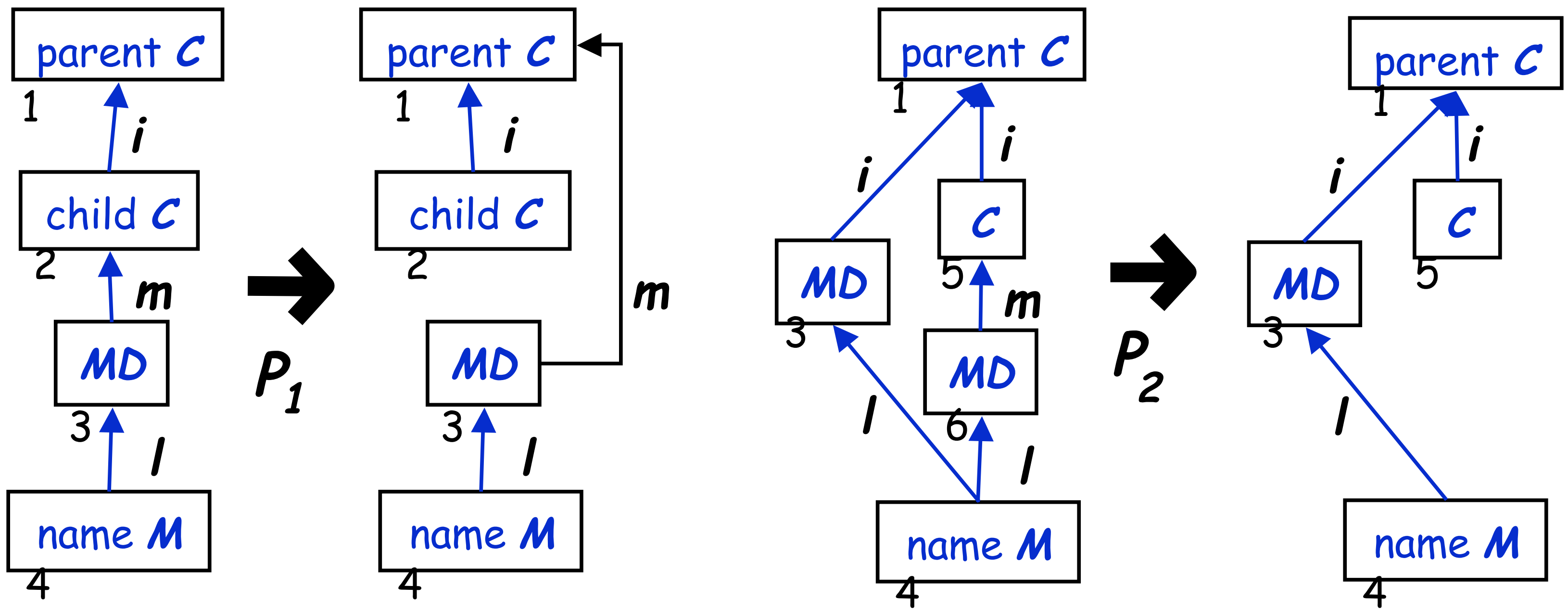
- Graph invariant

$$MD \xrightarrow{?*c} M \xrightarrow{/} MD$$

- preserves call behaviour
- Trivially fulfilled for *EncapsulateVariable*
  - all existing calls are *preserved*
  - *But: new calls are introduced for each variable access/update!*

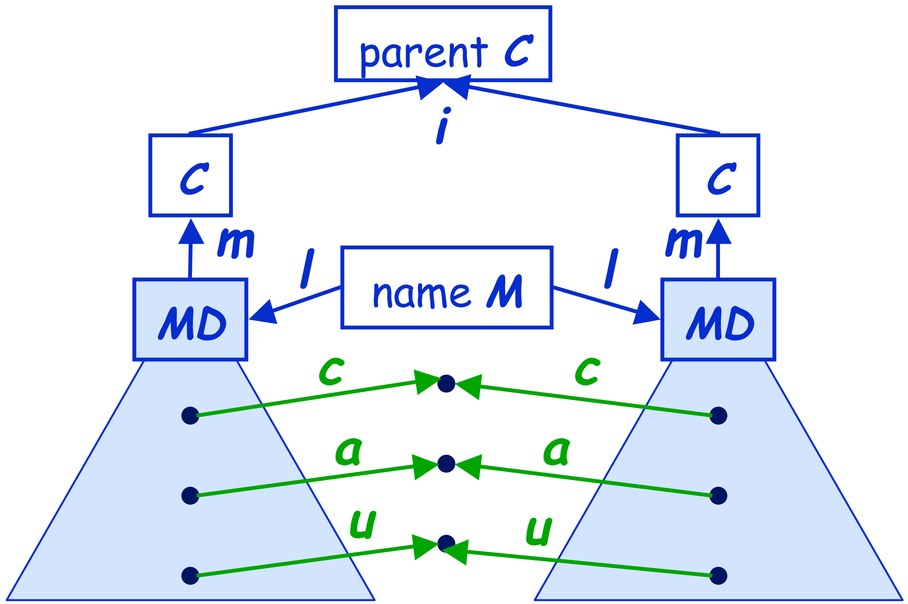
Formal experiment
Behaviour preservation

- PullUpMethod(parent,child,name)
  - affects all subclasses
  - need controlled graph rewriting



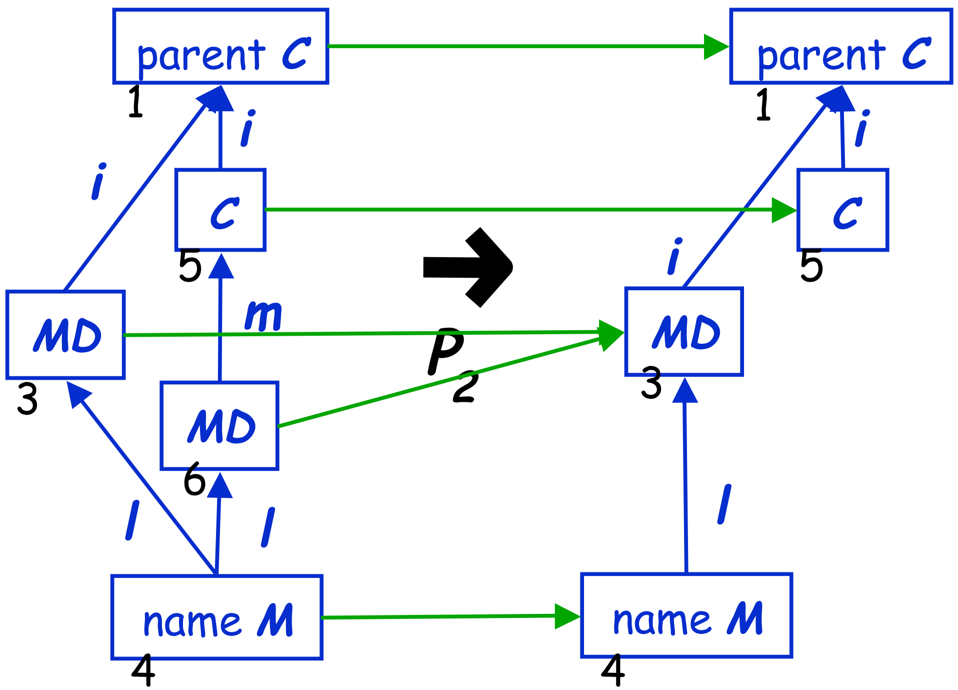
Formal experiment
Behaviour preservation

- *PullUpMethod* preserves calls, accesses, updates
  - *Only if we assume isomorphism between pulled up method definitions in subclasses*



Formal experiment
Behaviour preservation

- *PullUpMethod* preserves calls, accesses, updates
  - *Need tracking function to express method equivalence*



Formal experiment
Summary

- Initial results promising, but …
- Need to understand what is
  - behaviour
  - behaviour preservation
- Different notions of behaviour require different preservation properties
  - real-time systems (time constraints)
  - embedded systems (power & memory consumption)
  - safety critical systems (liveness, …)
- What are good program invariants ? How to express them ?

Tutorial outline

✓Introduction

✓Graph transformation theory

✓Graph transformation experiments

- Conclusion

Conclusions

Conclusion

- This tutorial has
  - Briefly introduced the theory of GT
  - Presented two state-of-the-art GT tools
  - Motivated the use of GT in software engineering
  - Reported on three concrete experiments to apply GT for model refactoring

Conclusion

- Graph transformation is *useful* for specifying refactorings
  - Language independent
  - Visual, flexible, precise representation
  - Verifying different kinds of behaviour preservation

Conclusion

- Graph transformation is *feasible* for specifying refactorings
  - Powerful GT engines exist, with state-of-the-art support for GT
  - Critical pair analysis in AGG
  - Story diagrams in Fujaba

  - Initial experiments show that ideas can be made independent of particular CASE tool

Conclusion

Graph Transformation	Refactoring
type graph, graph expressions	wf-constraints
graph invariants	behaviour preservavtion
negative application conditions (NAC)	preconditions
parameterised tranformation with embedding mechanism	refactoring transformation
attributes and attribute conditions	
controlled graph rewriting (Fujaba)	to compose primitive transformations and to control their order of application
critical pair analysis (AGG)	to detect parallel evolution conflicts

Future work

- Apply GT ideas to *other types of models*
  - sequence diagrams, statecharts, activity diagrams
- Compare theory of GT with *other theories and formalisms*
  - Description logics
  - Model checking
  - …
- Support *co-evolution* between models (of all kinds and at all levels) and source code
  - inconsistency management, traceability, change propagation, impact analysis

Future work

- Increase *efficiency* and *scalability* of GT tools
  - Use relational database technology as underlying engine for GT
    - Varró *et al.*
  - Use logic fact base to store graphs, and logic programming language to perform and reason about GT
    - JTransformer, Contract, Condor (based on Prolog)
      - Guenter Kniesel, University of Bonn
    - Description logics (e.g., RACER)
      - Ragnhild Van Der Straeten, Vrije Universiteit Brussel

Future work

- Use graph transformation to assist with other aspects of refactoring
  - (de)composition of refactorings
  - analyse complexity of refactorings
  - *triple graph grammars* to deal with co-evolution of refactorings at different levels

More questions

- How can we build more open refactoring tools?
- How can we determine where and why to refactor?
- Where does refactoring fit in the software development process?
- How to assess the effect of refactoring on software quality?
- How is refactoring related to other techniques ?
  - design patterns, application frameworks, aspect-oriented programming, generative programming, …

Further reading

- Handbook of Graph Grammars and Computing by Graph Transformation, World Scientific, 1999
  - Foundations
  - Applications, Languages and Tools
  - Concurrency, Parallelism, and Distribution
- Tutorial Introduction to Graph Transformation: A Software Engineering Perspective
  - L. Baresi, R. Heckel
    Proc. 1st Intl. Conference on Graph Transformation (ICGT 2002), Barcelona, Spain
    Springer LNCS 2505
- Bibliography website
  http://www.informatik.uni-bremen.de/theorie//appligraph/bibliography.html

Further reading

- Journal articles
  - A survey of software refactoring
    - T. Mens, T. Tourwé
    - IEEE Transactions on Software Engineering, February 2004
  - Formalising refactorings with graph transformations
    - T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens
    - Journal of Software Maintenance and Evolution, July/August 2005
  - A formal approach to model refactoring and model refinement
    - R. Van Der Straeten, T. Mens, V. Jonckers
    - Software and System Modeling, July 2005. *Conditionally accepted.*
  - Analysing refactoring dependencies using graph transformations
    - T. Mens, G. Taentzer, O. Runge
    - Software and System Modeling, July 2005. *Conditionally accepted.*

Further reading

- Book chapters
  - Using Graph Transformation for Practical Model Driven Software Engineering
    - L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, D. Varró
    - Model-driven Software Development - Volume II of Research and Practice in Software Engineering, edited by Sami Beydeda and Volker Gruhn, July 18, 2005. ISBN: 3-540-25613-X
- Websites
  - www.planetmde.org
    - "Everything you always wanted to know about MDE but were afraid to ask…"

Further reading

- Conference articles
  - Supporting model refactorings through behaviour inheritance consistencies
    - R. Van Der Straeten, V. Jonckers, T. Mens
    - Proc. UML 2004, LNCS 3273
  - Using description logics to maintain consistency between UML models
    - R. Van Der Straeten, T. Mens, J. Simmonds, V. Jonckers
    - Proc. UML 2003, LNCS 2863
  - Towards automating source consistent UML refactorings
    - P. Van Gorp, H. Stenten, T. Mens, S. Demeyer
    - Proc. UML 2003. LNCS 2863
  - Formalising behaviour preserving program transformations
    - T. Mens, S. Demeyer, D. Janssens
    - Proc. ICGT 2002. LNCS 2505