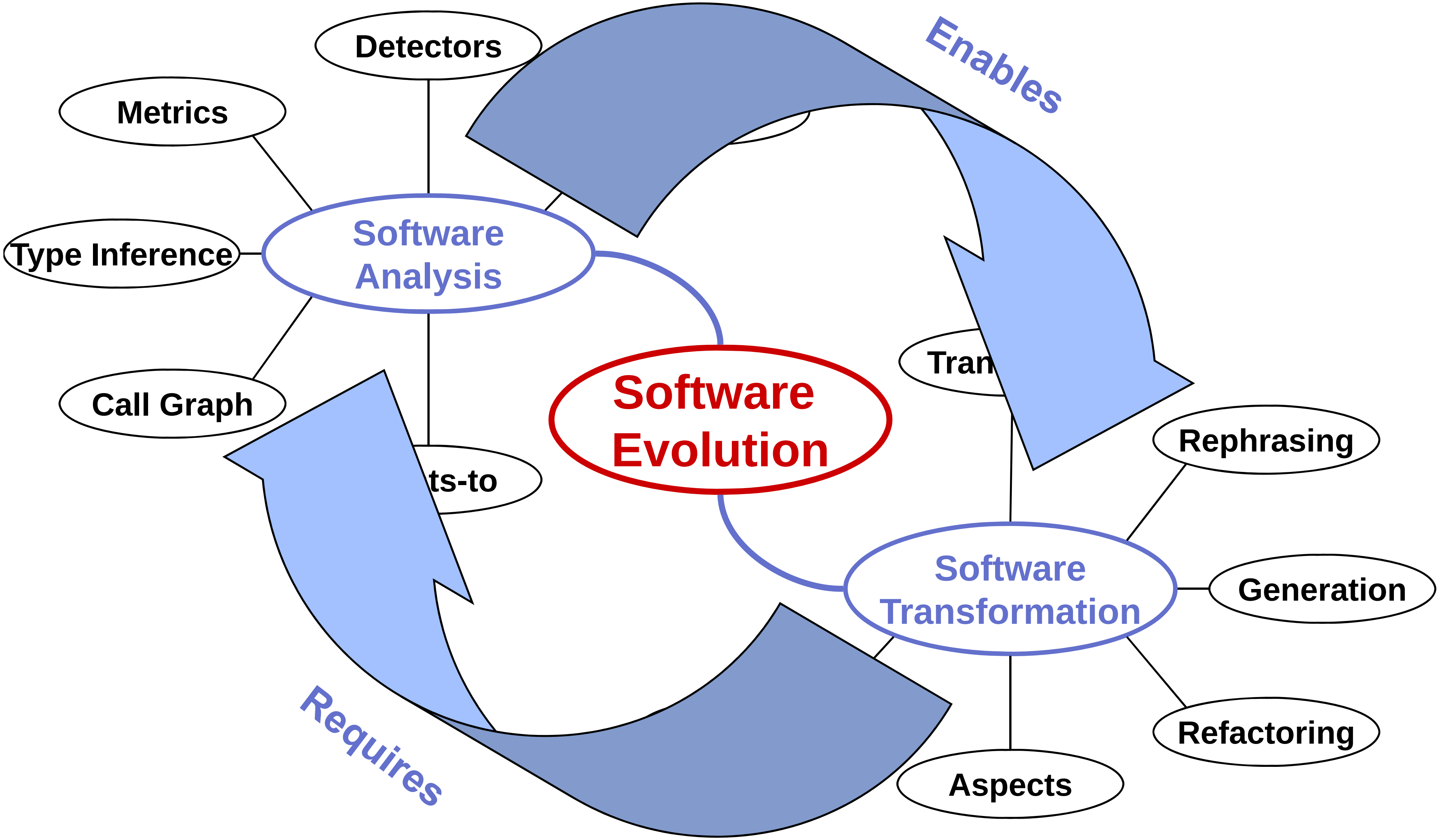
JTransformer

Logic-based Software-Analysis and Transformation for Java

Günter Kniesel, Tobias Rho
Alexis Raptarchis, Patrick Rypalla, Jan-Paul Imhoff

ROOTS Group
Computer Science Department III
University of Bonn

gk@cs.uni-bonn.de


Detectors
Metrics
Type Inference
Software Analysis
Call Graph
ts-to
Enables
Software Evolution
Tran
Rephrasing
Software Transformation
Generation
Refactoring
Aspects
Requires

Goal

- Integration
  - Uniform environment for software analysis and transformation

Additional Requirements

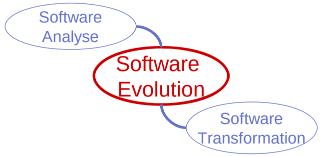
- Simplicity
  - Focus on what to do, not how → declarative

- Fast turn-around
  - Rapid prototyping, fast development
  - High run-time performance

- Scalability
  - Seconds, even on 1.000.000 LOC and beyond

- Usability
  - Smooth integration in development workflows

Overview



Approach

- Logic based Software Artefact Representation
- Logic-based Software Analysis
- Logic based Conditional Transformations (CTs)
- JTransformer: Logic-based Analysis and Transformation for Java

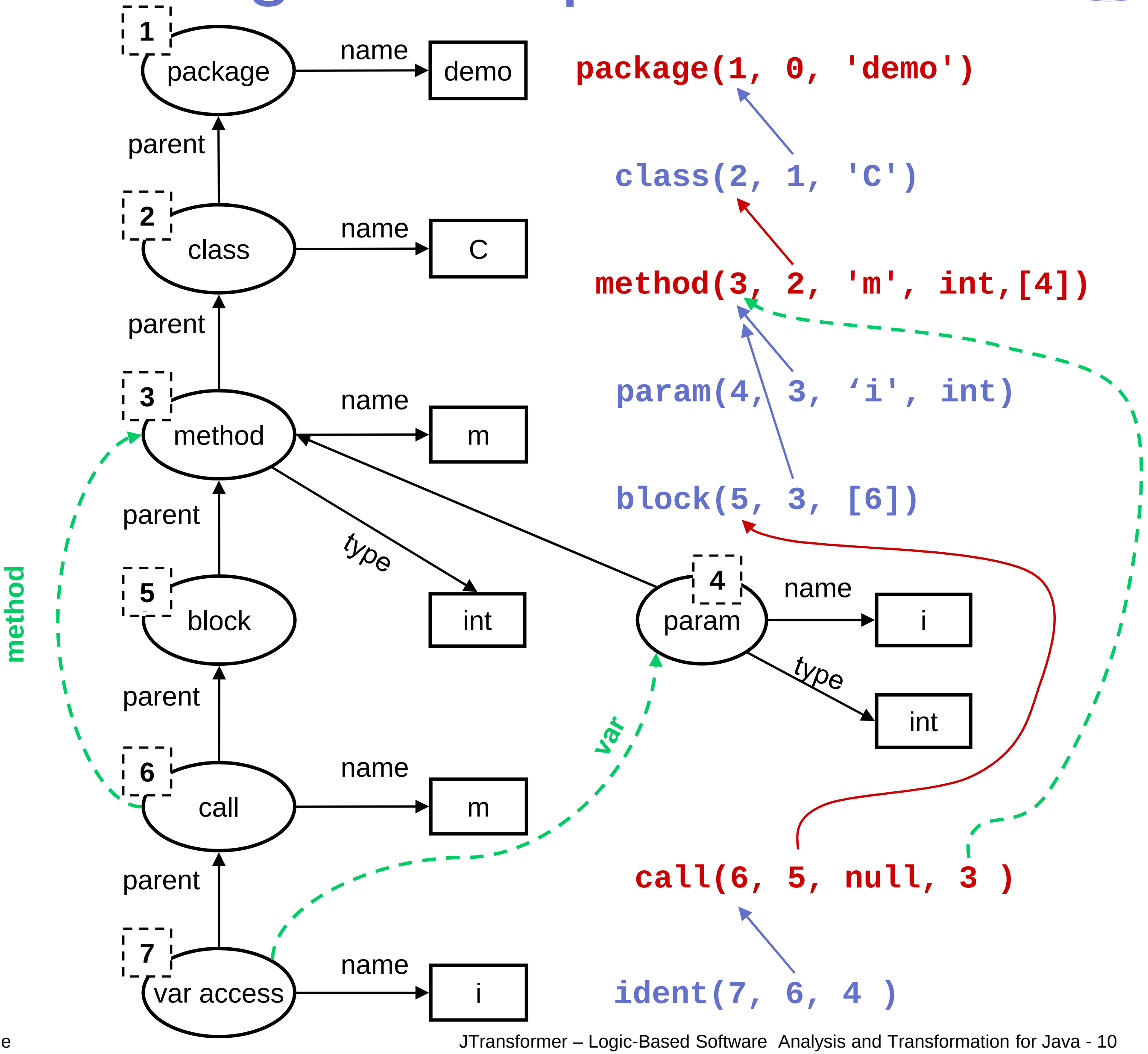
Case Studies

- Metrics and Smells
- Architecture Analysis
- Performance Analysis
- Smells and Refactorings

Logic-Based Software Representation

Logic-Based Program Representation

```
package demo;


class C {


    int m(int i) {




        m(i);
    }
}
```



package(1, 0, 'demo')

class(2, 1, 'C')

method(3, 2, 'm', int,[4])

param(4, 3, ‘i', int)

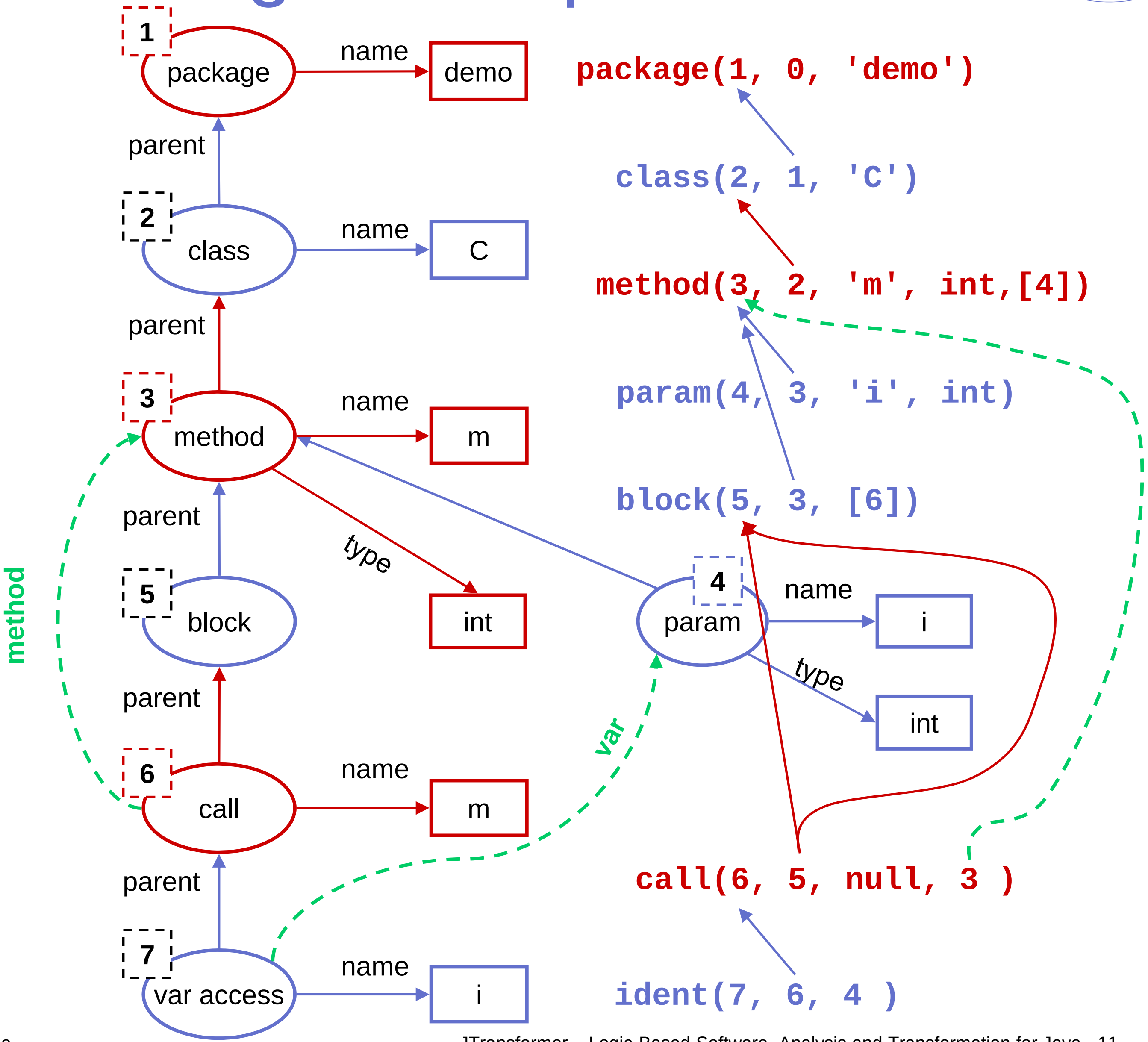
block(5, 3, [6])

call(6, 5, null, 3)

ident(7, 6, 4)

Logic-Based Program Representation

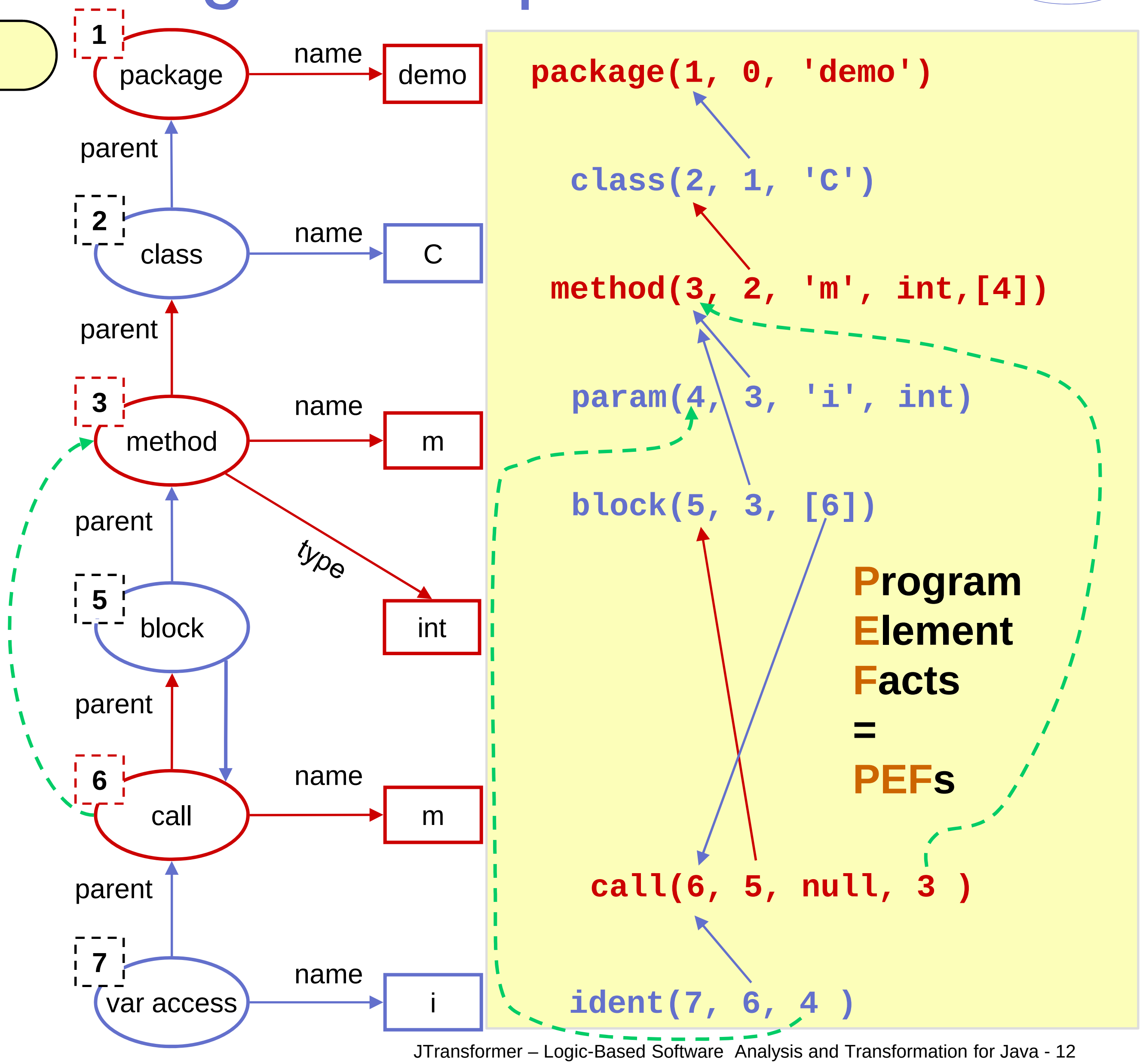
```
package demo;

class C {

   int m(int i) {

      m(i);
   }
}
```



```
package(1, 0, 'demo')
class(2, 1, 'C')
method(3, 2, 'm', int,[4])
param(4, 3, 'i', int)
block(5, 3, [6])
call(6, 5, null, 3 )
ident(7, 6, 4 )
```

Logic-Based Program Representation

Logic terms encode arbitrary graphs!



```
package(1, 0, 'demo')
  class(2, 1, 'C')
 method(3, 2, 'm', int,[4])
  param(4, 3, 'i', int)
  block(5, 3, [6])
   call(6, 5, null, 3 )
  ident(7, 6, 4 )
```

Program Element Facts = PEFs

Program Element Facts (PEFs)

- Complete representation of Java 1.4 Abstract Syntax Tree
  - Projects, files and packages
  - Interface elements (types and their members)
  - Code elements (statements and expressions)
  - Comments (javadoc and block comments)

- Representation of Java 1.5 / 1.6 Abstract Syntax Tree
  - Annotations
  - Syntactic suggar (foreach, …)
  - Generics: Work in progress (JTransformer 2.8 ++)
    - ➔ JT 2.8.0: Programs containing generics can be processed but no PEFs for generic type informations are created

JTransformer

- Eclipse Plug-In
  - Automatic creation of PEFs for Java projects
  - Incremental update of PEFs when source code is changed
  - Program analyses and transformations
    - ➔ Development environment
    - ➔ Run-time environment
  - Reverse engineering of Java source from transformed PEFs

- See http://sewiki.iai.uni-bonn.de/research/jtransformer

Logic-Based Software Analysis

Analysis-Examples

- Metrics
  - Anything you want → „Cultivate Plugin“ (Daniel Speicher)
- Architecture analysis
  - Dependencies, cycles, architectural rule enforcement
- Performance analysis
  - Smelly Database access patterns
- Bad Smells
  - Hints about need for refactorings
- Refactoring Preconditons
  - Type-Constraints, hierarchy structure, ...
- Code comprehension and Mining
  - Design Patterns, Crosscutting Concerns

Learning by Doing

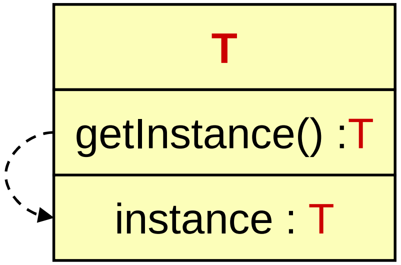
- Install JTransformer from memory sticks

- First steps
  - Open prepared workspace
  - Create factbase for JHotDraw project
  - Open PEF Documentation
  - Open Prolog Console

- Simple queries
  - Find a class: classDefT(Class, Parent, Name, Members)
    - ➔ Logic variables, conjunction, backtracking
  - Find a source class: … , not(externT(Class))
    - ➔ Comma means conjunction

© 2008-2010 Guenter.Kniesel@cs.uni-bonn.de

Logic-based Program Analysis Example: Pattern Mining



Singleton Pattern

A static method with 0 arguments in *Type* returns an instance of *Type* by accessing a static field in *Type* that has type *Type*!

```
classMethodReturnsOwnInstance(Type, Method, Field) :-

   methodDefT(Method, Type, _, [], type(_, Type, 0), _, _),
   modifierT(Method, static),

   fieldDefT(Field, Type, type(_, Type, 0),_,_),
   modifierT(Field, static),

   getFieldT(_, _, Method, _, _, Field).
```

- Query: "?- `classMethodReturnsOwnInstance(Type, Method, Field)`."
- Returns tuples of values for <`Type`, `Method`, `Field`> that represent singletons.
- Generates all results via backtracking.

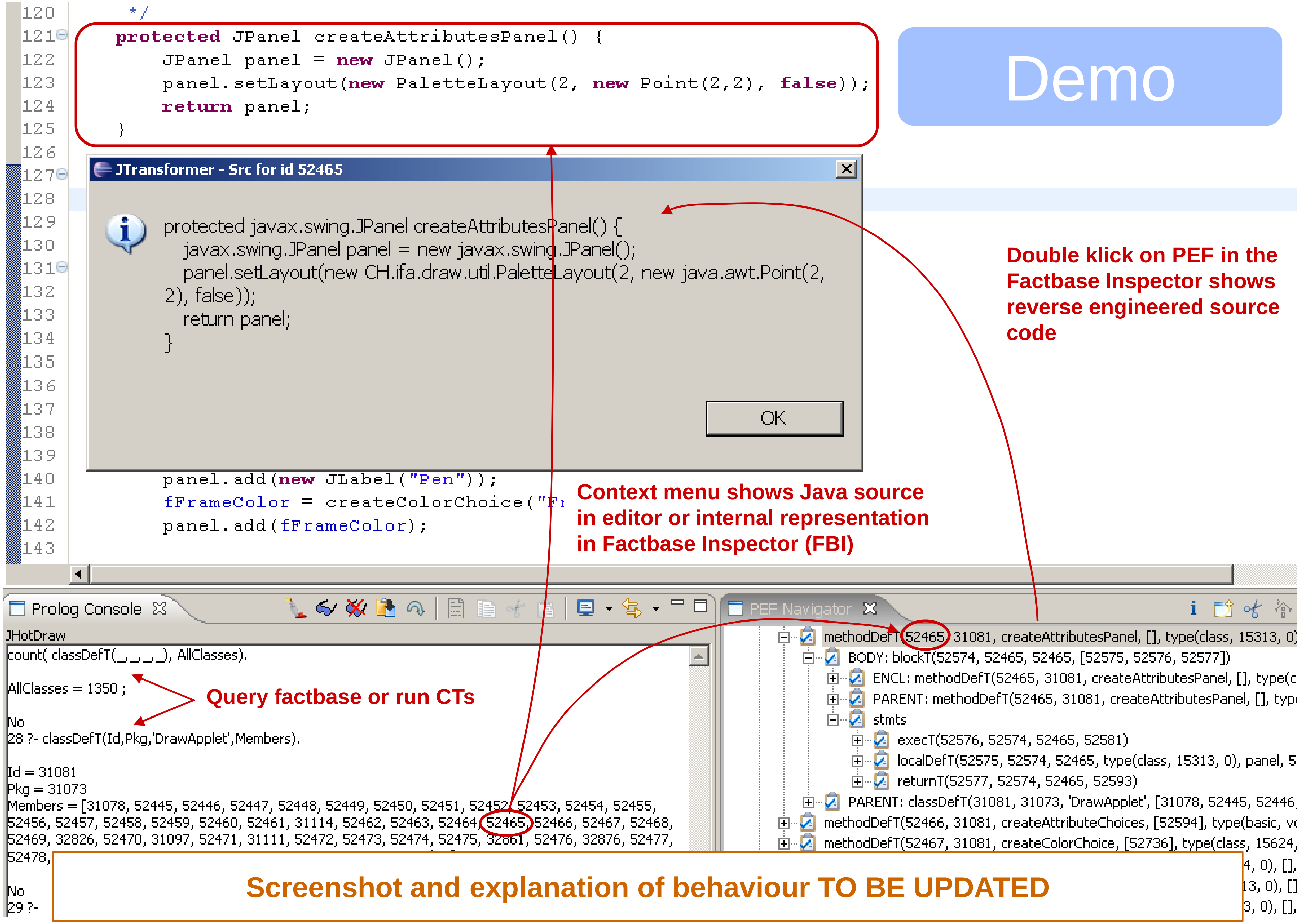
Learning by Doing

- Open JT-Tutorial/patterns/singleton.pl

- Run it
  - ?- `classMethodReturnsOwnInstance(Type, Method, Field).`

- Inspect the results using the multi-way linking feature

- Multi-way linking (Query – Source – Factbase)
  - Link console → editor
  - Link console → factbase
  - Link factbase → editor
  - Link editor → factbase



```
   */
protected JPanel createAttributesPanel() {
    JPanel panel = new JPanel();
    panel.setLayout(new PaletteLayout(2, new Point(2,2), false));
    return panel;
}
```

```
protected javax.swing.JPanel createAttributesPanel() {
  javax.swing.JPanel panel = new javax.swing.JPanel();
  panel.setLayout(new CH.ifa.draw.util.PaletteLayout(2, new java.awt.Point(2,
2), false));
  return panel;
}
```

```
    panel.add(new JLabel("Pen"));
    fFrameColor = createColorChoice("F
    panel.add(fFrameColor);
```

```
JHotDraw
count( classDefT(_,_,_,_), AllClasses).

AllClasses = 1350 ;

No
28 ?- classDefT(Id,Pkg,'DrawApplet',Members).

Id = 31081
Pkg = 31073
Members = [31078, 52445, 52446, 52447, 52448, 52449, 52450, 52451, 52452, 52453, 52454, 52455,
52456, 52457, 52458, 52459, 52460, 52461, 31114, 52462, 52463, 52464, 52465, 52466, 52467, 52468,
52469, 32826, 52470, 31097, 52471, 31111, 52472, 52473, 52474, 52475, 32861, 52476, 32876, 52477,
52478,

No
29 ?-
```

```
⊟ methodDefT(52465, 31081, createAttributesPanel, [], type(class, 15313, 0)
  ⊟ BODY: blockT(52574, 52465, 52465, [52575, 52576, 52577])
    ⊞ ENCL: methodDefT(52465, 31081, createAttributesPanel, [], type(c
    ⊞ PARENT: methodDefT(52465, 31081, createAttributesPanel, [], typ
    ⊟ stmts
      ⊞ execT(52576, 52574, 52465, 52581)
      ⊞ localDefT(52575, 52574, 52465, type(class, 15313, 0), panel, 5
      ⊞ returnT(52577, 52574, 52465, 52593)
  ⊞ PARENT: classDefT(31081, 31073, 'DrawApplet', [31078, 52445, 52446,
⊞ methodDefT(52466, 31081, createAttributeChoices, [52594], type(basic, vo
⊞ methodDefT(52467, 31081, createColorChoice, [52736], type(class, 15624,
4, 0), [],
13, 0), []
3, 0), [],
```

Demo

Double klick on PEF in the Factbase Inspector shows reverse engineered source code

Context menu shows Java source in editor or internal representation in Factbase Inspector (FBI)

Query factbase or run CTs

Screenshot and explanation of behaviour TO BE UPDATED

More Analysis Examples

Metrics: Depth of Inheritance (DOI)

Find out the inheritance depth of any class.

Learn to write recursive predicates.

```
metric_doi(Class, 0) :-
   classDefT(Class, _, _, _),
   not(extendsT(Class,_Super)).

metric_doi(Class, DOI) :-
   classDefT(Class, _, _, _),
   extendsT(Class, Super),
   metric_doi(Super, DOISuper),   // ← recursive call
   DOI is DOISuper + 1.
```

- Predicate defined by multiple clauses → Disjunction
- Recursion in the second clause.

Metrics: Depth of Inheritance

Write a smell detector that uses the metric.

Should only report results for source classes.

```
smell_doi(Limit, DOI,FQN):-
   metric_doi(Class,DOI),          % use the metric
   not(externT(Class))             % for source classes only
   DOI >= Limit,                   % metric has critical value
   fullQualifiedName(Class,FQN).    % get full name of class (JT)


?- smell_doi(5, DOI, Type).        % Run the detector
```

- Try it out on JHotDraw.

Architecture: Package Cycles

1. Write a detector for type-level dependencies.

Start by considering the causes / reasons for such dependencies.

```
/**
 * depBetweenTypes(?DependingClass, ?Reason, ?ReferencedClass)
 *
 * Calculate dependency between types including their reasons.
 * Reasons can be
 *   - an extension or implementation of the other type
 *   - a field, method, parameter or local variable declaration
 *     that uses the other type in its signature
 *   - an access to a field or invocation of a method declared
 *     in the other type
 */
```

- This is what we want.
- Now let‘s do it.

Architecture: Package Cycles

1. Write a detector for type-level dependencies.

Start by considering the causes / reasons for such dependencies.

```
depBetweenTypes(DependClass, extends, RefClass) :-
       extendsT(DependClass, RefClass) .

depBetweenTypes(DependClass, hasfield(FieldDef), RefClass) :-
    fieldDefT(FieldDef, DependClass, type(class,RefClass,_), _, _).

depBetweenTypes(DependClass, hasparam(Param), RefClass) :-
    paramDefT(Param, Method, type(class,RefClass,_), _),
    methodDefT(Method, DependClass, _,_,_,_,_).

depBetweenTypes(DependClass, calls(Call,Method), RefClass ) :-
    callT(Call,_,CallingMethod,_,_,_,Method),
    methodDefT(CallingMethod,DependClass,_,_,_,_,_),
    methodDefT(Method,RefClass,_,_,_,_,_).
```

- Other cases follow the same pattern
  - see „JT-Tutorial/architecture/typeDependencies.pl“

Architecture: Package Cycles

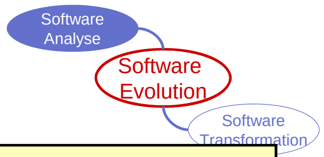
1. Write a detector for type-level dependencies.

→ Remove self-dependencies

```
dependencyBetweenTypes(DepClass, Reason, RefClass) :-
   depBetweenTypes(DepClass, Reason, RefClass),
   not(DepClass = RefClass).
```

- Other cases follow the same pattern
  - see „JT-Tutorial/architecture/typeDependencies.pl“

Architecture: Package Cycles



1. Write a detector for type-level dependencies.

→ Store the results and group them by dependent types.

```
deriveTypeDependencies :-
    deriveAndStoreRawTypeDependencies,
    groupTypeDependenciesByReason.

deriveAndStoreRawTypeDependencies :-
    retractall( rawTypeDependency(_,_,_) ),
    forall(
       dependencyBetweenTypes(DepClass,Reason,RefClass),
       assert( rawTypeDependency(DepClass,Reason,RefClass) )
   ).
```

- **`assert`** and **`retract`** add / remove facts from the Prolog database
  - **`retractall`** retracts all facts that match

Architecture: Package Cycles

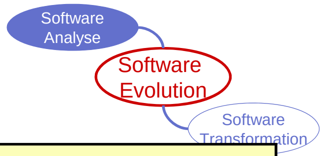
1. Write a detector for type-level dependencies.

→ Store the results and group them by dependent types.

```
deriveTypeDependencies :-
     deriveAndStoreRawTypeDependencies,
     groupTypeDependenciesByReason.

groupTypeDependenciesByReason :-
   retractall( groupedTypeDependency(_,_,_) ),
   rawTypeDependency(DepClass, _, RefClass),
     not( groupedTypeDependency(DepClass, RefClass, _) ),
     findall( Reason,
              rawTypeDependency(DepClass, Reason, RefClass),
              Reasons
     ),
     assert( groupedTypeDependency(DepClass, RefClass, Reasons) ),
   fail.
groupTypeDependenciesByReason.
```

- „Loop via backtracking“ is a typical Prolog idiom

Architecture: Package Cycles



2. Write a detector for package-level dependencies.

→ Remove self-dependencies

```
derivePackageDependencies :-
    deriveAndStoreRawPackageDependencies,
    groupPackageDependenciesByReason.

deriveAndStoreRawPackageDependencies :-
    retractall( rawPackDependency(_,_,_) ),
    forall(
       dependencyBetweenPackages( DepPack, RefPack, Reason ),
       assert( rawPackDependency( DepPack, RefPack, Reason ) )
   ).
```

- Other cases follow the same pattern
  - see „JT-Tutorial/architecture/typeDependencies.pl“

Architecture: Package Cycles

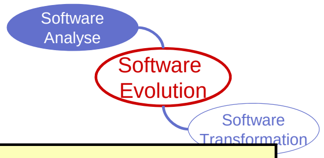
2. Write a detector for package-level dependencies.

→ Remove self-dependencies

```
derivePackageDependencies :-
    deriveAndStoreRawPackageDependencies,
    groupPackageDependenciesByReason.

dependencyBetweenPackages( DepPackage, RefPackage, Reason ) :-
    groupedTypeDependency(DepClass, RefClass, Reasons),
    not(externT(DepClass)),
    not(externT(RefClass)),
    class_in_package(DepPackN,DepClsN,DepPackage,DepClass),
    class_in_package(RefPackN,RefClsN,RefPackage,RefClass),
    not(DepPackage = RefPackage),
    Reason = classdependency(DepClass, RefClass, Reasons).
```

- Other cases follow the same pattern
    - see „JT-Tutorial/architecture/typeDependencies.pl“

Architecture: Package Cycles



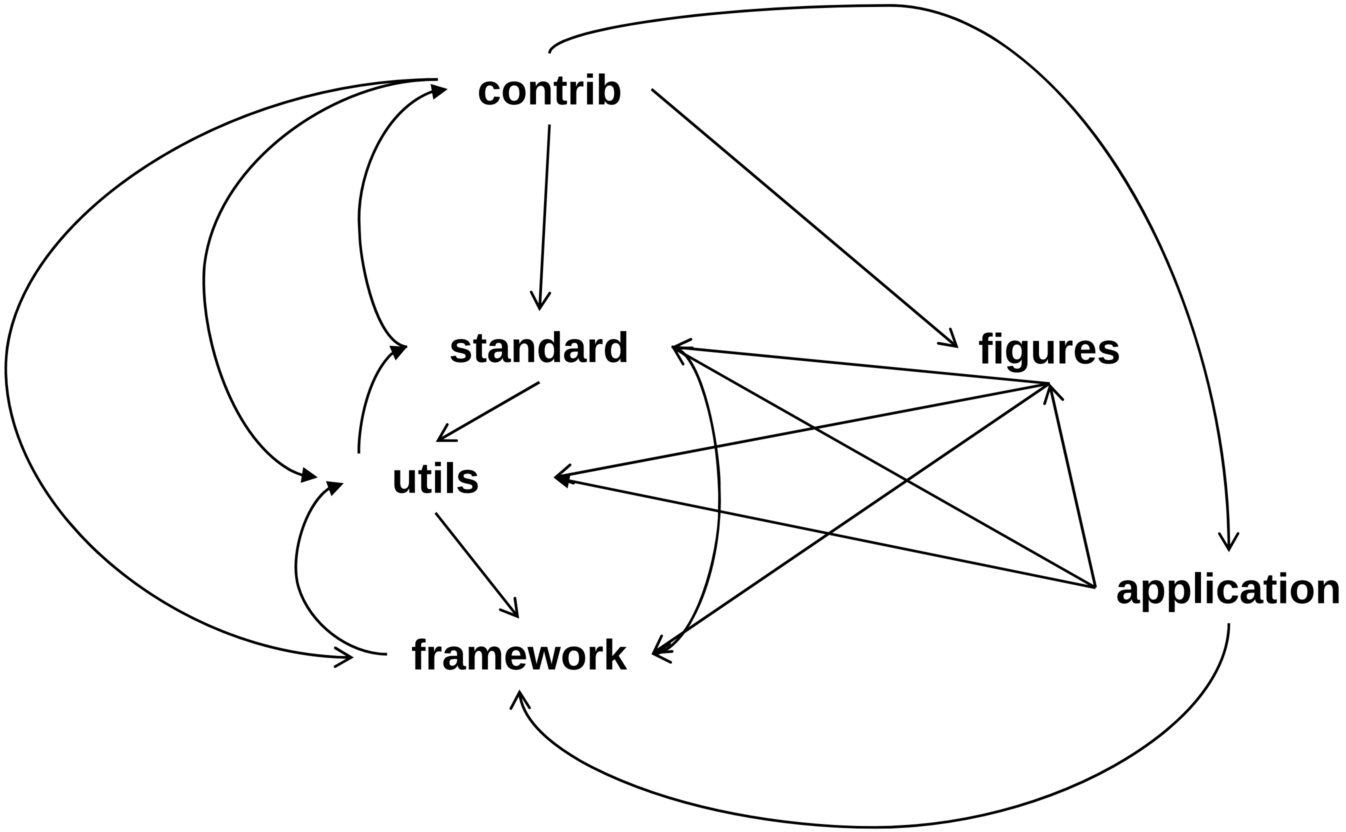
3.Put everything together.

```
typeAndPackageDependencies :-
    deriveTypeDependencies,                 % stores dependencies
    derivePackageDependencies,              % stores dependencies
    find_package_cycles(PackageCycles),     % Tarski algorithm
    find_class_cycles(ClassCycles),         % Tarski algorithm
    report_results(PackageCycles,ClassCycles).

?- typeAndPackageDependencies.
Found
  22233 raw dependencies between different types
   7965 non-redundant type dependencies
     41 non-redundant package dependencies
      1 non-redundant package dependency cycles
     42 non-redundant type-level dependency cycles.
```

- Let‘s try it!

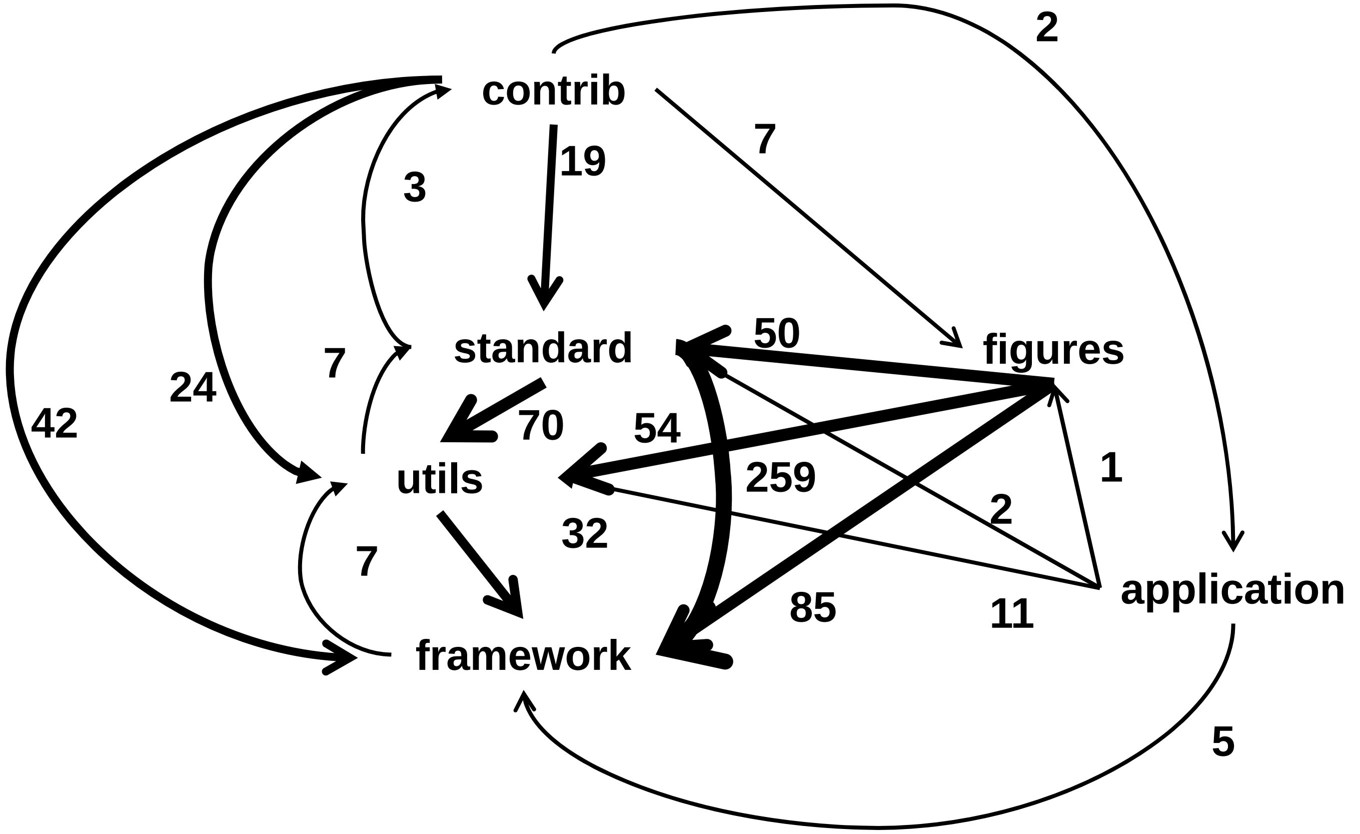
JHotDraw 6.01b Package Dependencies

- Use the information about the reasons for dependencies
  - Graph weihted by number of reasons for a dependency



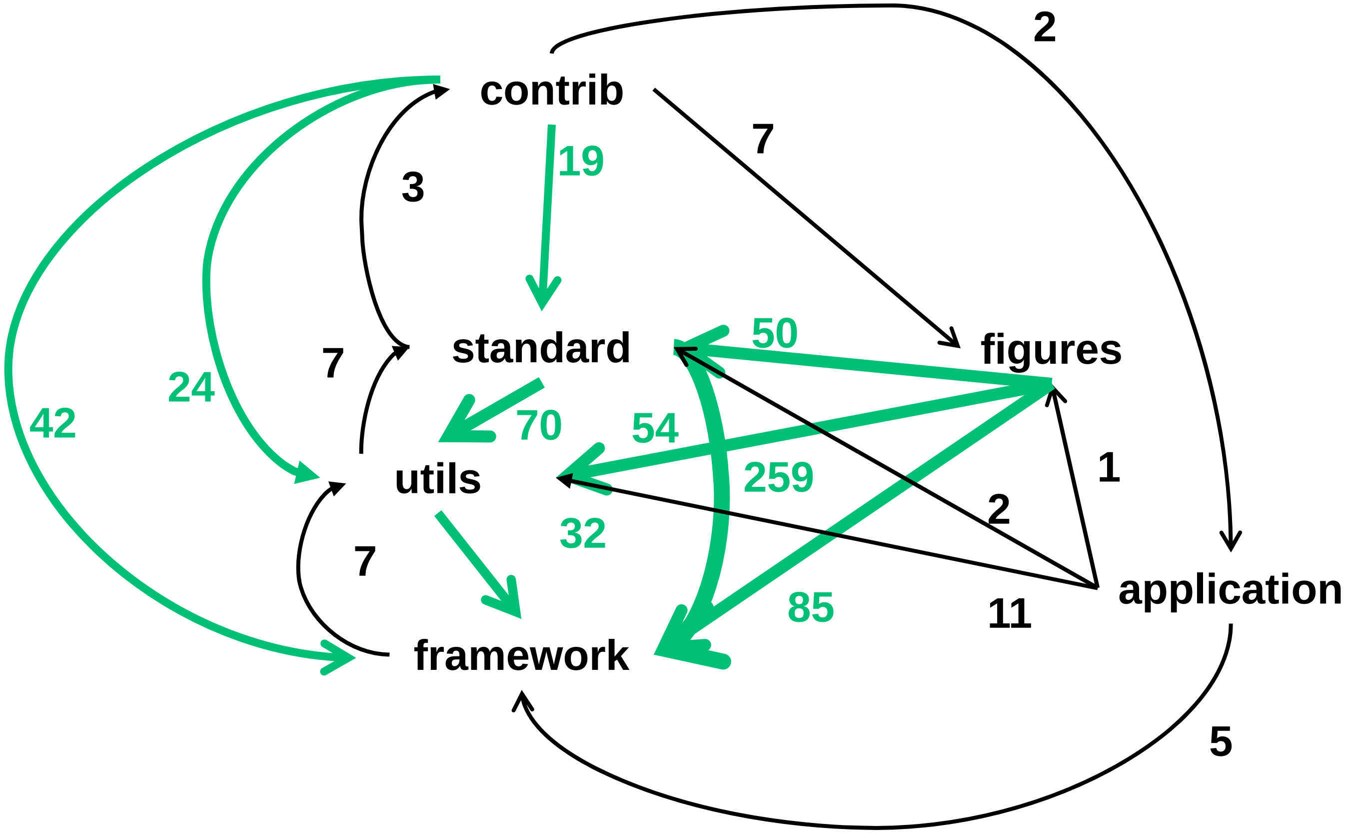
JHotDraw 6.01b Package Dependencies

- Use the information about the reasons for dependencies
  - Graph weihted by number of reasons for a dependency



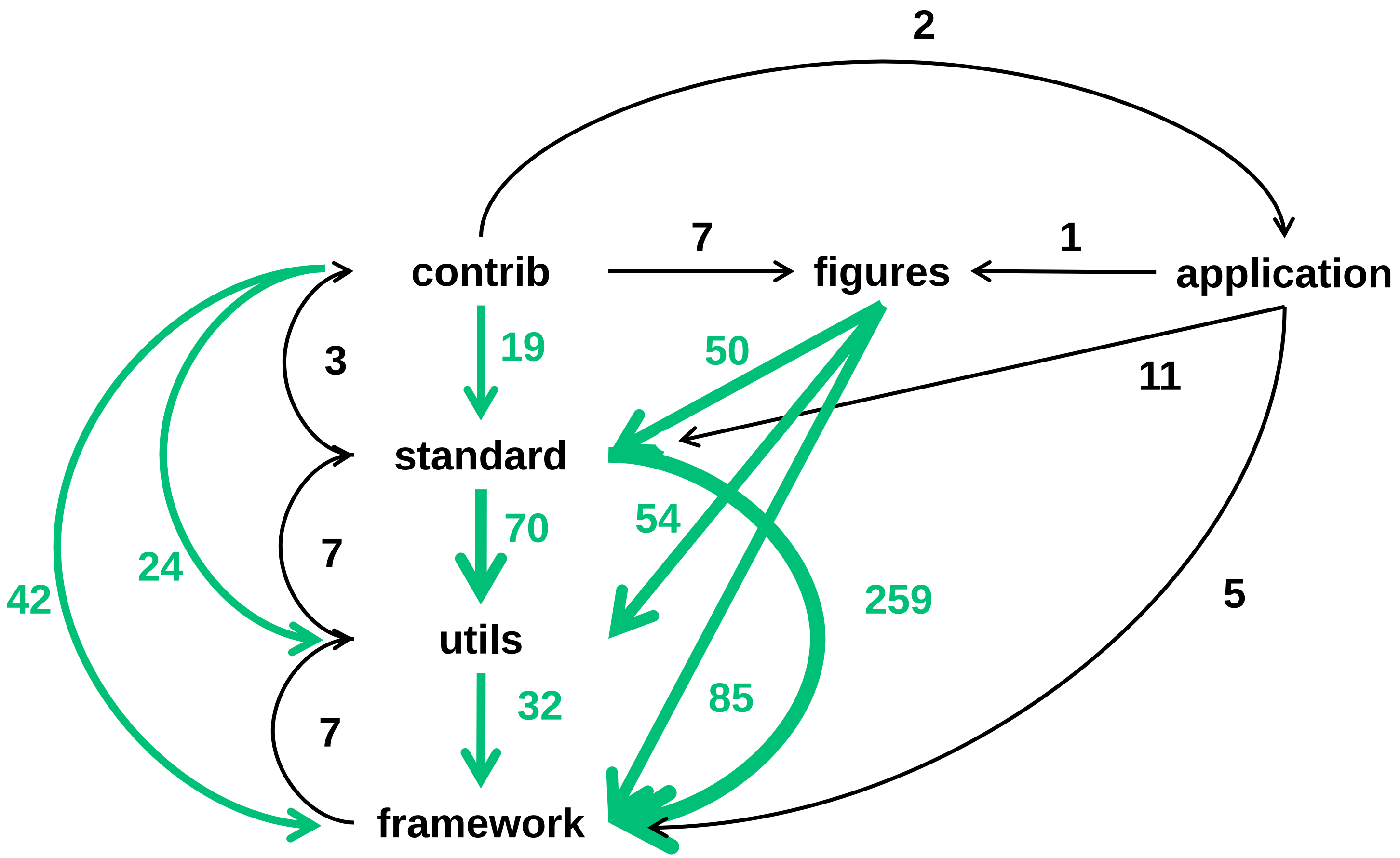
JHotDraw 6.01b Package Dependencies

- Color “heavy” edges (weight > 15) green
  - Not many chances to remove these



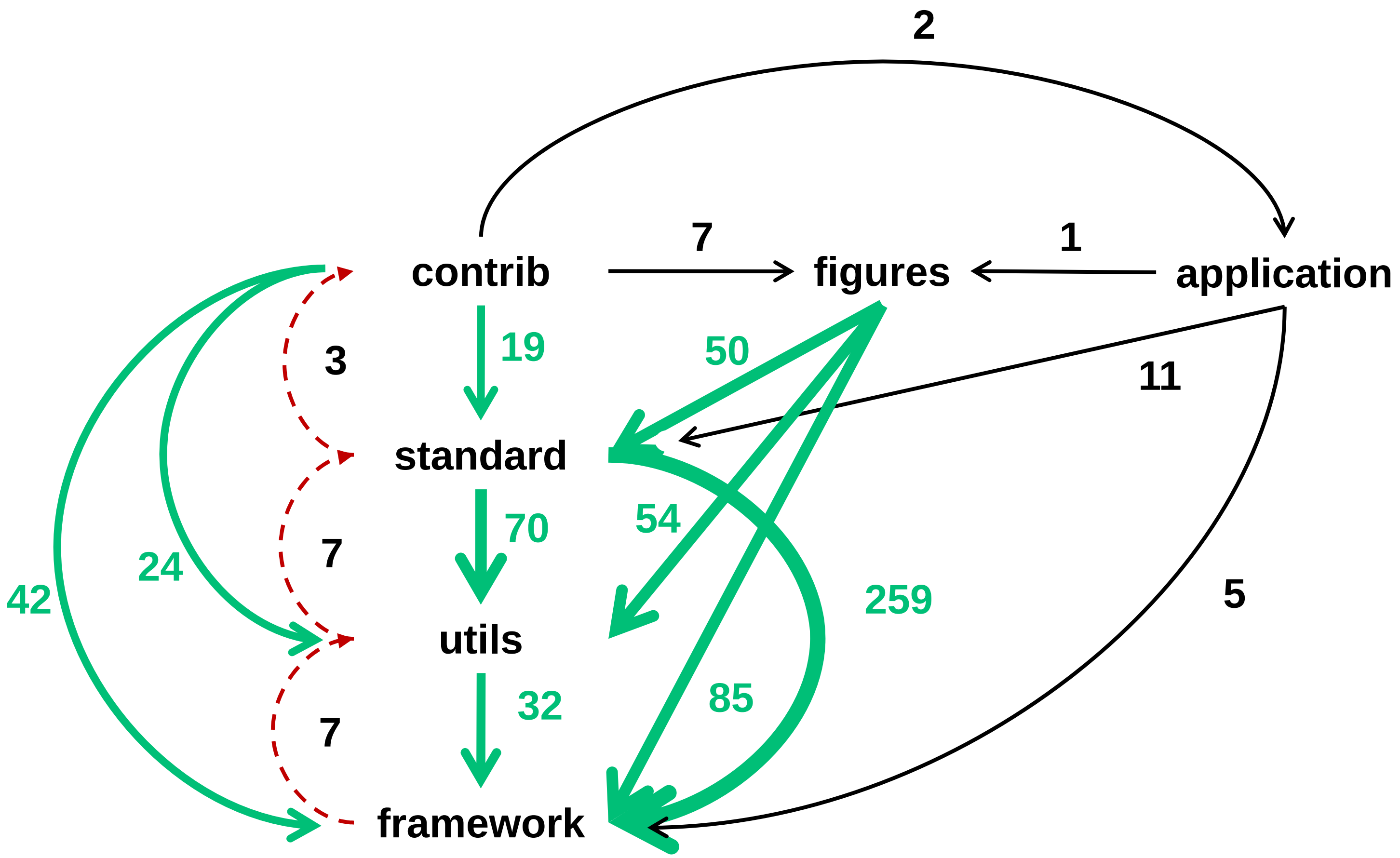
JHotDraw 6.01b Package Dependencies

- Restructure the graph to potential layers by directing green arrows downwards



JHotDraw 6.01b Package Dependencies

- Highlight minimal set of “easy” dependencies responsible for cycles



Performance Evaluation

– Program Representation and Analysis –

Factbase Creation Performance

- Case Study: Design Pattern Detection
  - Kniesel, Hannemann, Rho: “A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction”, Proc. of LATE’07 (Linking Aspect technology and Evolution), ACM.
- Scenarios
  - A) Initial factbase creation (incl. compiliation of Java projects)
  - B) Reload of saved factbase (textual format)
  - C) Reload of saved factbase (binary format)

Factbase Creation Performance (seconds)

Benchmark	**A) Initial Factbase**	**B) Text Reload**	**C) Binary Reload**
JHotDraw 6.0b1	20	< 0,5	< 0,05
Eclipse 3.1 Core	< 15*60	< 3*60	< 20

JTransformer 2.3
(June 2007)

Query Performance

- Case Study: Design Pattern Detection
  - Kniesel, Hannemann, Rho: “A Comparison of Logic-Based Infrastructures for Concern Detection and Extraction”, Proc. of LATE’07 (Linking Aspect technology and Evolution), ACM.
- Task
  - Find all instances of the “Observer” design pattern
- Scenarios
  - A) Naïve Prolog Query
  - B) Optimized Query

Performance: Find All Observers

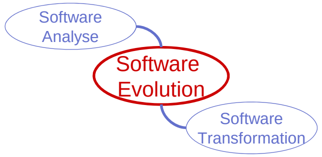
Benchmark	**A) Prolog**	**B) Optimized**
JHotDraw 6.0b1	> 3 hours	0,06 sec
Eclipse 3.1 Core	---	< 8,00 sec

JTransformer 2.3
(June 2007)

Analysis and Transformation

© 2008-2010 Guenter.Kniesel@cs.uni-bonn.de

Overview



Approach

- Logic based Software Artefact Representation
- Logic-based Software Analysis
- Logic based Conditional Transformations (CTs)
- JTransformer: Logic-based Analysis and Transformation for Java

Case Studies

- Metrics and Smells
- Architecture Analysis
- Performance Analysis
- Smells and Refactorings

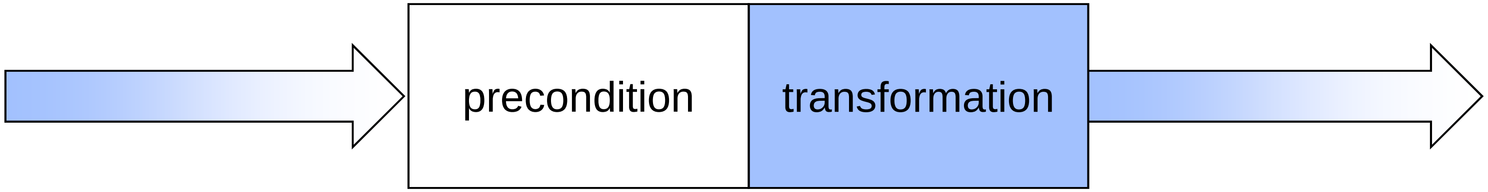
Bad Smells: Indicators of Refactoring needs

```
non_private_field(Class,Field,FieldType,FieldName,Modif) :-
    fieldDefT(Field,Class,FieldType,FieldName,_),
    modifierT(Field,Modif),
    ( Modif = public
    ; Modif = package
    ; Modif = protected
    ).


field_without_getter(Field,Class,Type,Name,Getter) :-
    non_private_field(Class,Field,Type,Name,_Modif),
    concat(get, Name, Getter),
    % No method with signature "Type Getter()" :
    not( methodDefT(_Meth,Class,Getter,[],Type,_,_) ).
```

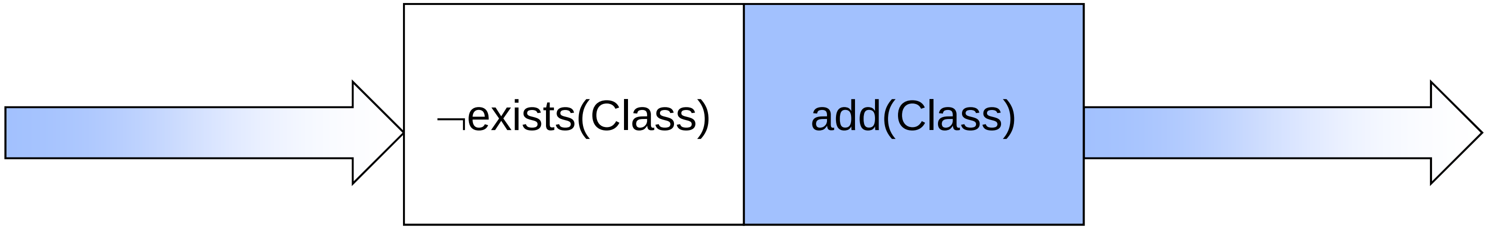
- Suppose, we find 231 non-encapsulated fields!
- Would you bother to encapsulate them?

Conditional Transformations (CTs)

- CT = Condition + Transformation
  - Condition is true ⇒ Transformation may be executed



- Example: „Add Class" Transformation
  - Condition: Class does not exist



Conditional Transformations with JT

Fixed set of Program Element Facts:

PEF $\in$ { classDefT(Id,Pkg,Name.Members),
fieldDefT(Id,Class, Name,Init), ...,
applyT(...), ... }

Condition Language

$C \rightarrow EC \wedge C \mid C \vee C \mid \text{not}(C)$

$EC \rightarrow \text{true} \mid \text{false} \mid \text{newId(Var)} \mid \text{pef} \in \text{PEF}$

Transformation Language

$\text{T} \rightarrow \text{ET} \mid \text{ET} \text{ , } \text{T}$

$\text{ET} \rightarrow \text{add(pef)} \mid \text{delete(pef)} \mid \text{replace}(\text{pef}_1,\text{pef}_2) \ : \text{pef} \in \text{PEF}$

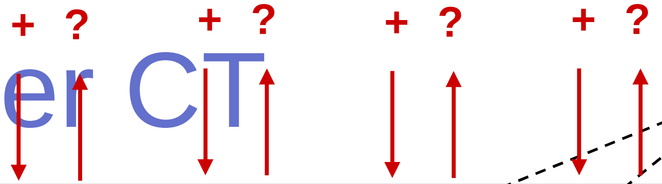
Create Accessor Method

- AddGetter CT
  - for all fields that have no getter method ...
  - ... add method that returns the field's value

```
public class C {

  B b = new B();





  ...

}
```

```
public class C {

  private B b = new B();

  B getB() {
     return b;
  }

   ...

}
```

AddGetter CT



No method with signature
"<Type> get<Name>()"
ists.

Head / Signature

```
ct( addGetter(Class,Field,Type,GName) ,  (
    classDefT(Class,_,_,_), not(exter
    fieldDefT(Field,Class,Type,Name,_),

    concat(get, Name, GName),
    not( methodDefT(Getter,Class, _Name,[],Type,_,_),
         getFieldT(_, _, Getter, _, _, Field) ),
    new_id(Method),...,new_id(Get)
),(
    add( methodDefT(Method,Class, GName,[],Type,[],Block) ),
    add( blockT(Block,Method,Method,[
    add( returnT(Return,Block,Method,Get) ),
    add( getFieldT(Get,Return,Method,null,Name,Field) ),
    add_to_class(Class,Method)
)).
```

Precondition

New identities for
new elements:

Transformation

Create method
"<Type> get<Name>() { return <Field>}":

Full „Encapsulate Field“ Refactoring

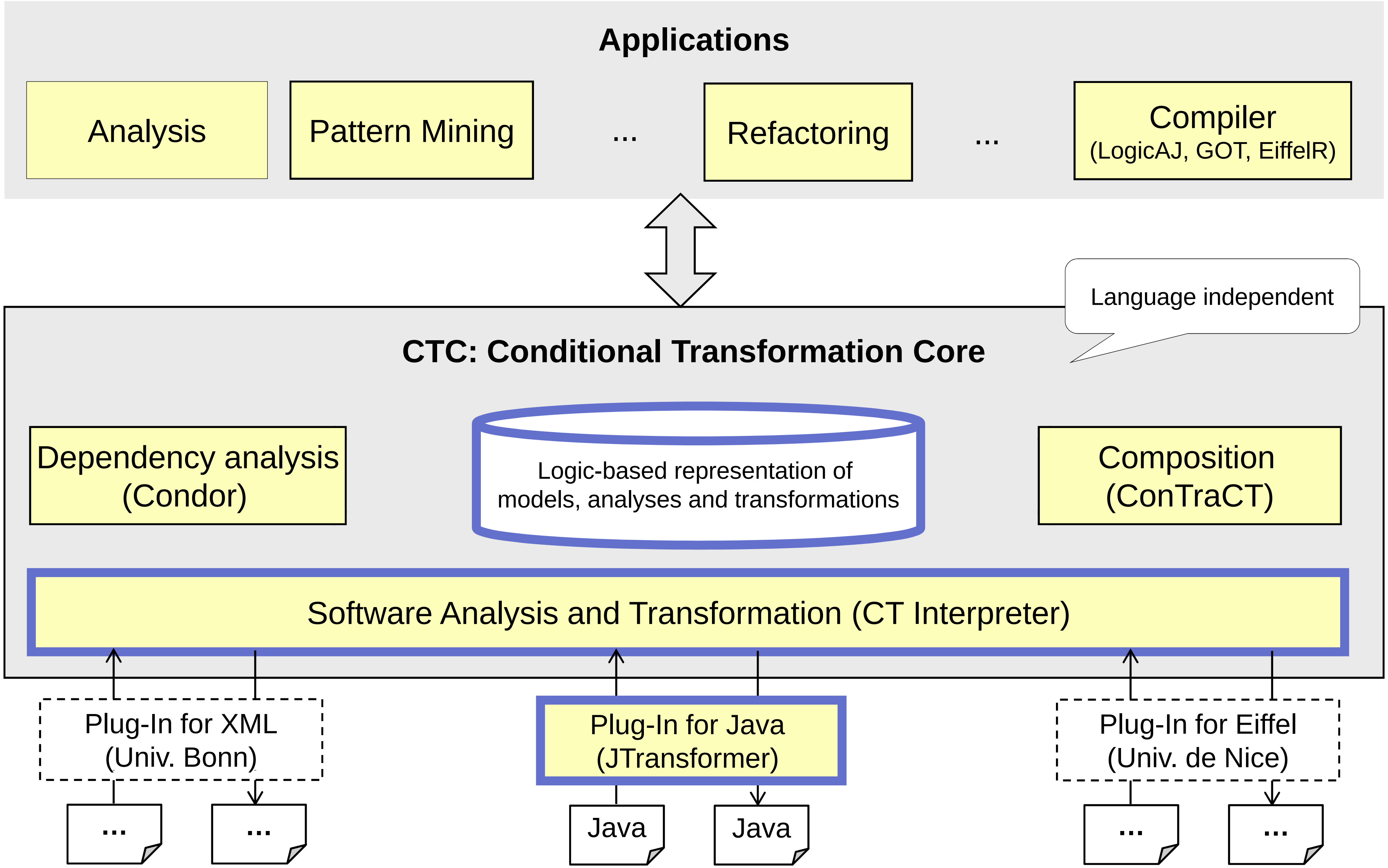
- Five CT definitions
  - AddGetter
  - AddSetter
  - ReplaceReadAccesses
  - ReplaceWriteAccesses
  - MakeFieldPrivate

- A CT sequence definition
  - invoke all of the above in the specified order

- Syntax of CT invocation in JTransformer
  - apply_ct(Head): invoke a CT
  - apply_ctlist([Head1, ..., HeadN]): invoke a sequence of CTs

Beyond Java – StarTransformer

Beyond Java: StarTransformer



Applications

Analysis

Pattern Mining

…

Refactoring

…

Compiler
(LogicAJ, GOT, EiffelR)

Language independent

CTC: Conditional Transformation Core

Dependency analysis
(Condor)

Logic-based representation of
models, analyses and transformations

Composition
(ConTraCT)

Software Analysis and Transformation (CT Interpreter)

Plug-In for XML
(Univ. Bonn)

Plug-In for Java
(JTransformer)

Plug-In for Eiffel
(Univ. de Nice)

Defining Your Own Language

- 1. Design Program Element Terms
  - The AST of your language
- 2. Specify Program Element Term structure in a standard form
  - The meta-model of your language
- 3. Implement a "Reader"
  - The parser of your language that generates Prolog facts conforming to the meta-model specified in 2.
- 4. Implement a "Writer"
  - The generator that converts PEFs into source code
- 5. (Optional): Provide a set of predefined conditions and CTs for analysing and transforming programs in your language

→ Plugin for StarTransformer that enables it to work with your language.

1. Defining Program Element Terms

- Syntax → Program Element Terms
  - Non-terminals are good candidates for PET types

- Determine attributes of PET types
  - Names, ...

- Determine relations between elements
  - e.g. extends(Sub,Super)

- Avoid mutual references
- Consider good database schema design rules
  - functional dependencies
  - normal forms (4NF)

2. Specification of PEF structure

- PET structure: fieldDefT(id#, parent#, type, name, expr#)
- Specified by:

language **PEF type (= AST node type)**

```
ast_node_def('Java',fieldDefT,[
    ast_arg( id,        mult(1,1,no ),  id,    [fieldDefT]   ),
    ast_arg( parent,    mult(1,1,no ),  id,    [classDefT]   ),
    ast_arg( type,      mult(1,1,no ),  attr,  [typeTerm]    ),
    ast_arg( name,      mult(1,1,no ),  attr,  [atom]        ),
    ast_arg( expr,      mult(0,1,no ),  id,    [...]  ),
    ast_arg( modifier,  mult(0,*,no ), attr,  [atom]        )
]).
```

specification of PEF arguments

argument name

multiplicity

order
(no = not ordered,
ord = ordered)

kind of value
(id = identity,
attr = primitve)

legal syntactic type(s) of argument values
(multiple types are allowed,
e.g. an expr, can have many)

3. Reader & 4. Writer

- Reader
  - Implement a parser
  - Possibly use an existing one and just implement a visitor on its internal AST that creates appropriate Prolog facts

- Writer
  - Can be implemented in Prolog, immediately creating source in your language
    - ➔ look at file 'java_writer.pl' for an example how this was done for Java
  - Alternatively you can write out the factbase to a Prolog text file
    - ➔ using the call 'writeTreeFacts(FileToWhichToWrite)'
  - ... and then use any tool you like to convert the generated text to your language syntax

State of Affairs

Done

- LogicAJ: Translation of a generic aspect language to CTs
- jTransformer: CT-based transformation engine for Java
- Condor: CT-based dependency analysis
- CTC: Logic-based core for language-parametric system

Ongoing

- StarTransformer: language-parametric transformation tool
  - Extract language-independent parts of JTransformer
  - Integrate them with the CTC and Condor
  - Develop language plugins with cooperation partners

→ http://roots.iai.uni-bonn.de/research/

→ http://sewiki.iai.uni-bonn.de/research/