Chapter 3.
Logic-based Software Analysis

Interactive Software Development Assistants

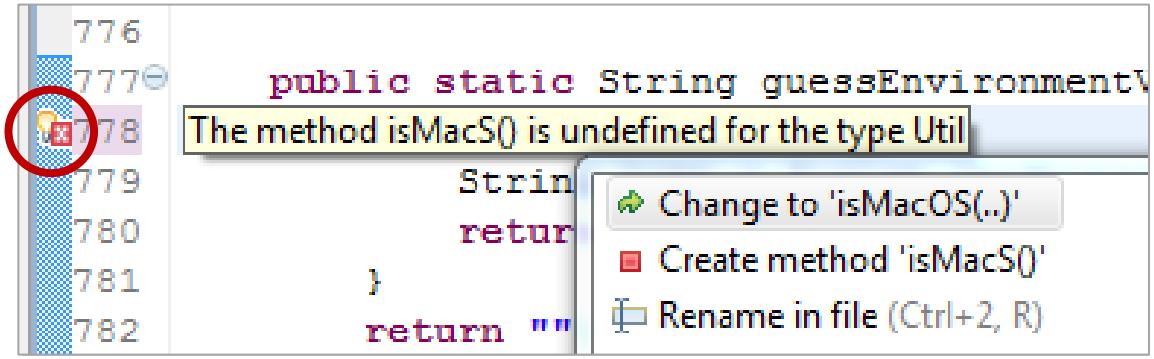
Logic-based Software Representation

Logic-based Software Analysis

Logic-based Software Transformation

Motivation
Software Development Assistants

- Example: „Monitor, advise, act“ cycle in the Eclipse IDE



- Behind the scenes: Software Analysis and Transformation (SAT)

- Research goal: Make SAT development quick and easy
  - Focus on conceptual essence of SAT tasks
  - … not on APIs, implementation, manual tuning, …

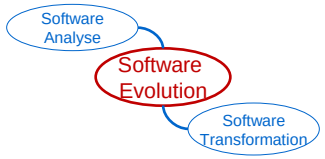
Goal

- Integration
  - Free, 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 into development workflows

Overview



Approach

- Logic based Software Artefact Representation
- Logic-based Software Analysis
- Logic based Software Transformation
- JTransformer: Logic-based Analysis and Transformation for Java

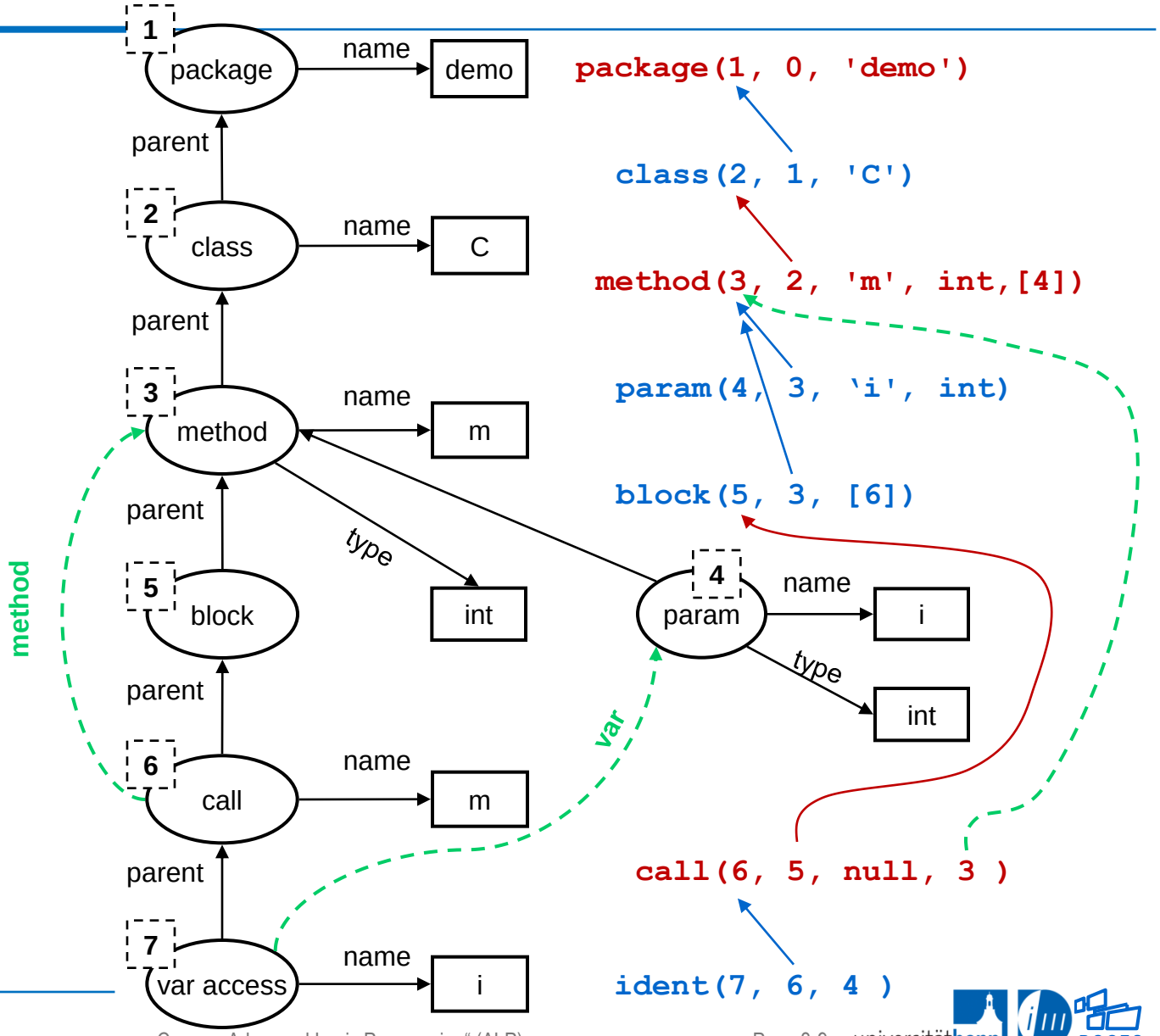
Case Studies

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

Logic-Based Software Representation

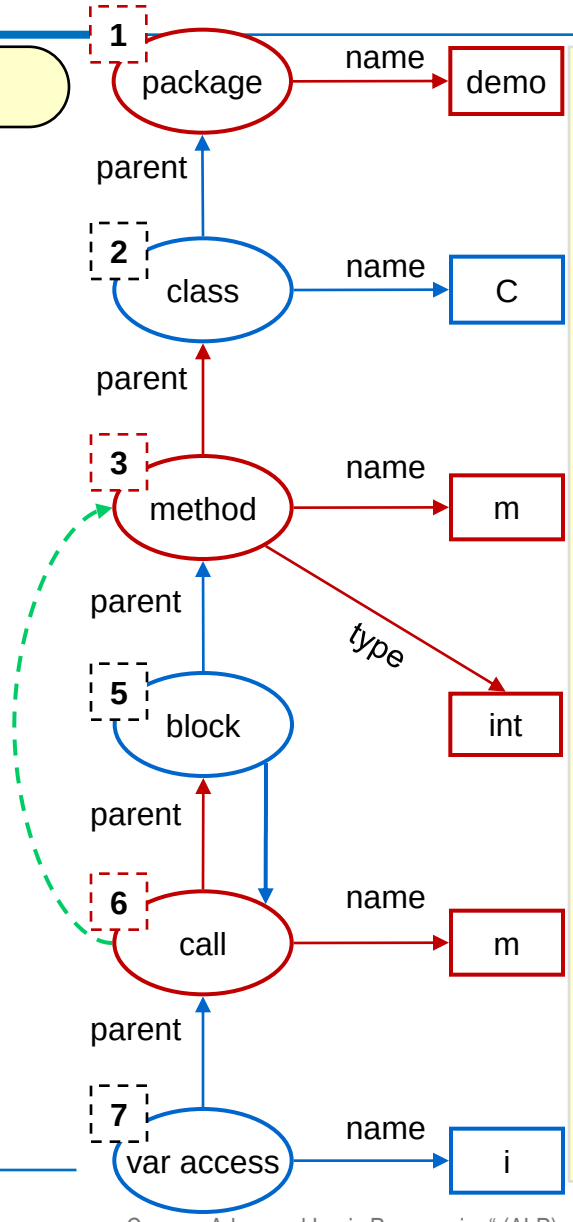
Logic-Based Program Representation

```
package demo;


class C {


    int m(int i) {


        m(i);
    }
}
```



Logic-Based Program Representation

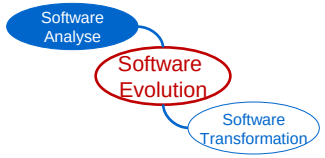
Logic terms encode arbitrary graphs!




```
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

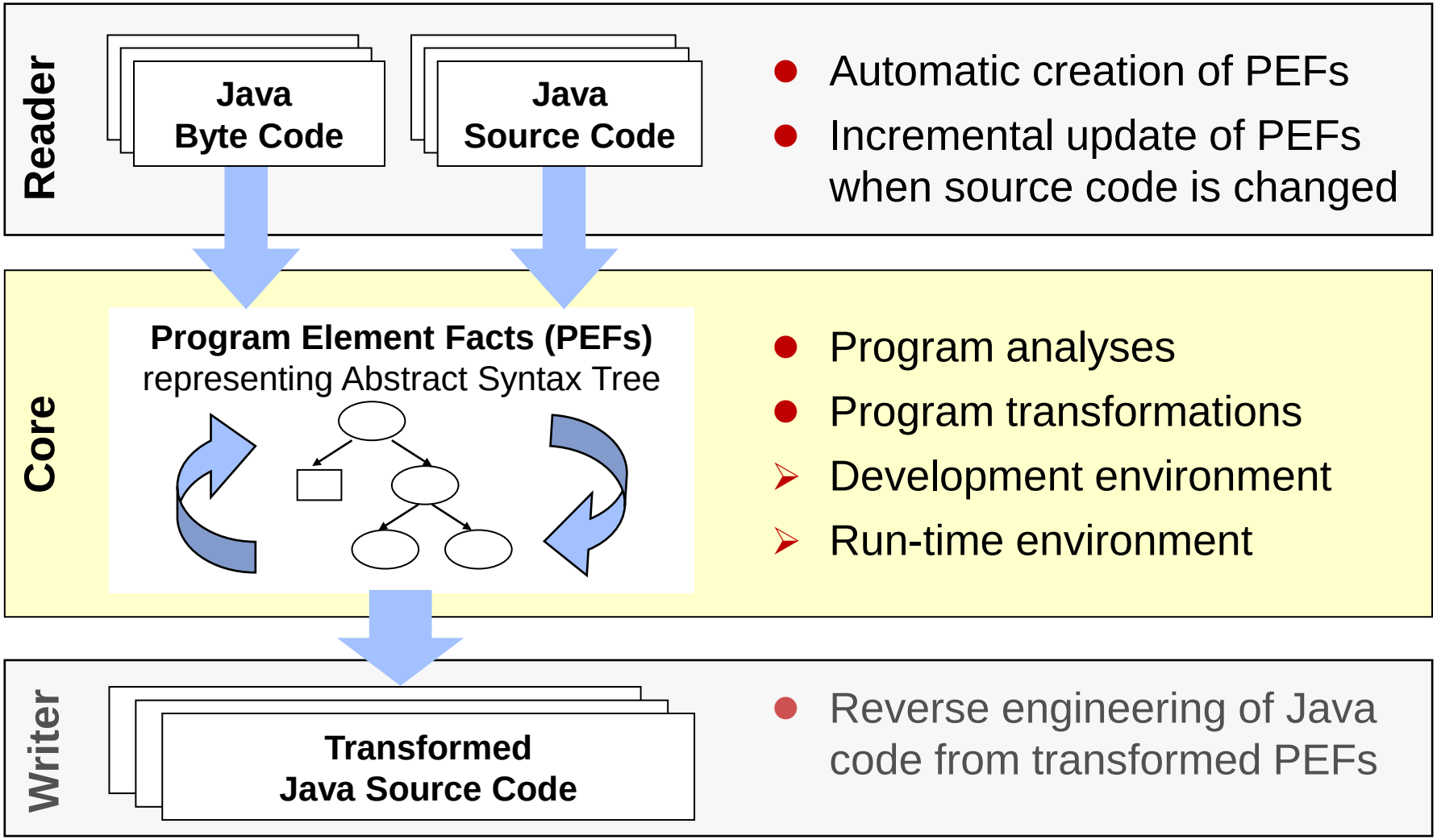
Program Element Facts (PEFs)



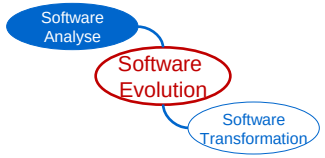
- Complete representation of Java 1.4 Abstract Syntax Tree
  - Projects, files and packages
  - Interface elements (types and their members)
  - Code elements (statements and expressions)
  - Comments (javadoc and block comments)

- 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



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
  - Installation
  - Tutorial
  - PEF Documentation

Learning by Doing

- Install JTransformer
  - See online Installation Guide
- First steps
  - Add „JHotDraw“ project to your workspace (or any project you want)
  - Create factbase for your project
  - Open PEF Documentation
  - Open Prolog Console
- Simple queries
  - Find a class:

```
?- classT(Class, Parent, Name, Members).
```

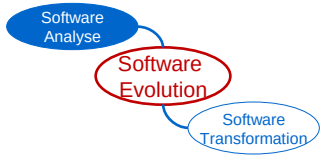
    - ⇨ Logic variables, backtracking
  - Find a source class:

```
?- classT(Class, Parent, Name, Members) , not(externT(Class)).
```

    - ⇨ Comma (',') means conjunction

Logic-Based Software Analysis

Analysis-Examples



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

Metrics: Depth of Inheritance (DOI)

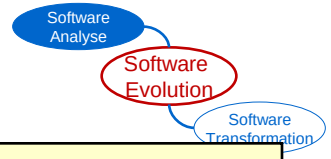
Find out the inheritance depth of any class.

Learn to write recursive predicates.

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

metric_doi(Class, DOI) :-
    classT(Class, _, _, _),
    extendsT(Class, Super),
    metric_doi(Super, DOISuper),    // recursive call
    DOI is DOISuper + 1.            // evaluation of arithm. expr.
```

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

Metrics: Depth of Inheritance



Write a smell detector that uses the previous metric.

Should only report values above a certain limit 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, FQN).              % Find classes at depth >= 5.
```

- Try it out on JHotDraw.

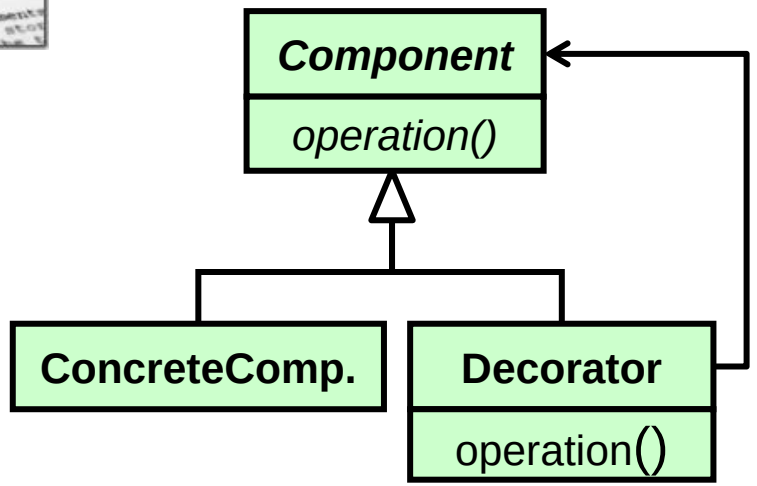
Bad Smells: Indicators of Refactoring needs

```
non_private_field(Class,Field,FieldType,FieldName,Modif) :-
    fieldT(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( methodT(_Meth,Class,Getter,[],Type,_,_) ).
```

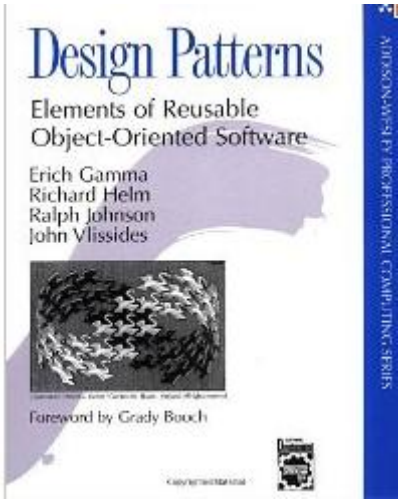
- Suppose, we find 231 non-encapsulated fields!
- Would you bother to encapsulate them one by one?

Design Pattern Detection

reconstruct
DESIGN



understand design
INTENTION



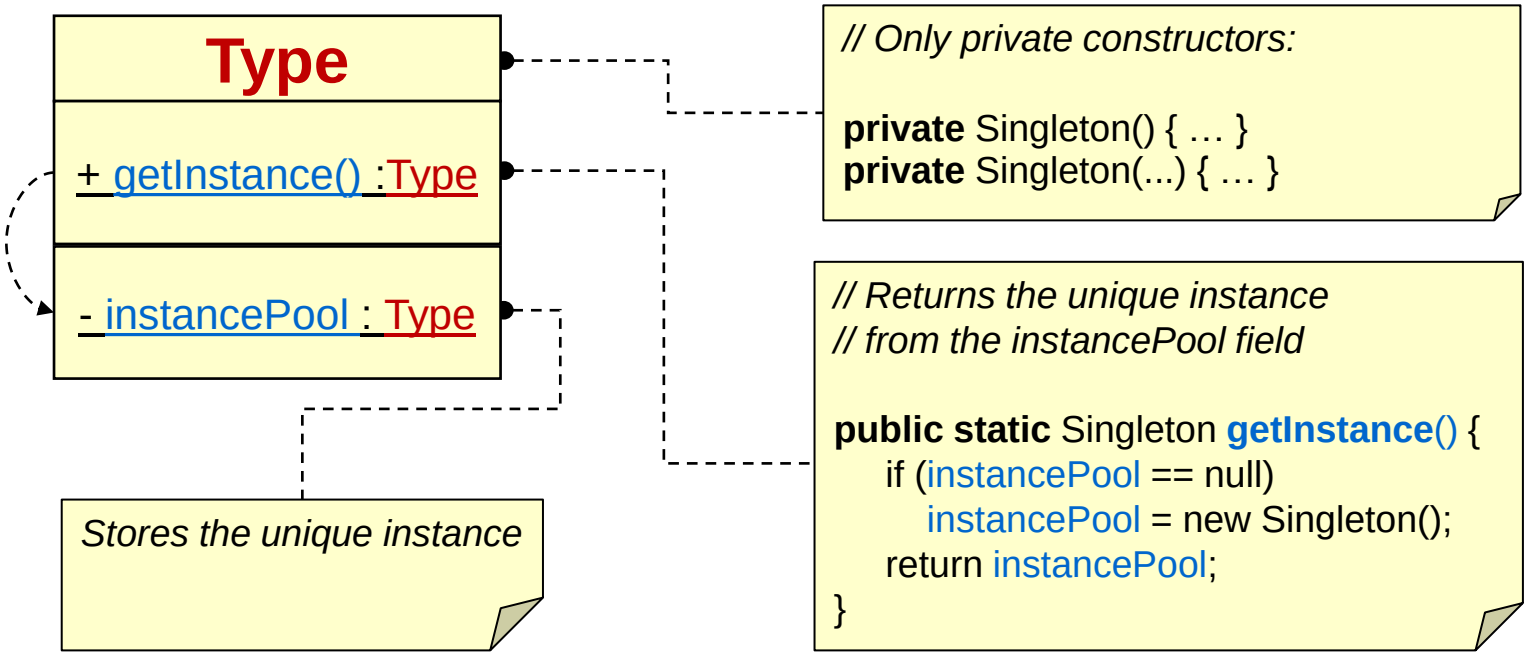
Singleton Pattern

Singleton Intention: Limit the number of instances of a class

- Typically: Just one instance
  - Motivation: Central access point
  - E.g. Facade, Repository, System, Abstract Factory
- Also: Fixed number of instances
  - Motivation 1: limited resources.
  - Motivation 2: avoid expensive object creation by „Object Pool“
    → e.g. create 1000 Enterprise Java Beans, use when needed, put each back into pool after use

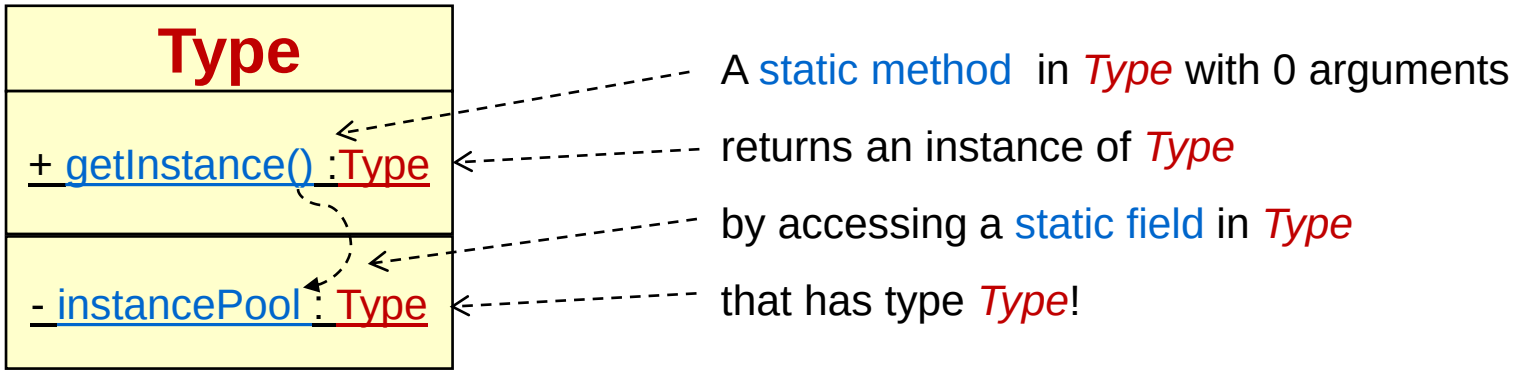
Singleton Pattern

Singleton Structure + Implementation



Singleton Pattern Detection

Singleton Structure + Implementation



Singleton Detector Predicate

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

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

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

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

Singleton Pattern Mining

Singleton Detector Predicate

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

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

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

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

Running the Singleton Detector

Query ?- staticMethodReturnsOwnInstance(Type, Method, Field).

- Returns tuples of values for <Type, Method, Field> that represent singletons.
- Generates all results via backtracking.

Learning by Doing

1. Generate the JTransformer-Example project
   - Eclipse → „New“ → „Example“ → „JTransformer“
2. Open JT-Tutorial/patterns/singleton.pl
3. Consult it (F9)
4. Run the query
   - ?- `classMethodReturnsOwnInstance(Type, Method, Field).`
5. Inspect results using multi-way linking (Query – Source – Factbase)
   - Prolog console → Source editor
   - Prolog console → Factbase inspector
   - Source editor ←→ Factbase inspector → Reengineered source

Multi-Way Linking

```
120     */
121    protected JPanel createAttributesPanel() {
122        JPanel panel = new JPanel();
123        panel.setLayout(new PaletteLayout(2, new Point(2,2), false));
124        return panel;
125    }
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140        panel.add(new JLabel("Pen"));
141        fFrameColor = createColorChoice("Fr
142        panel.add(fFrameColor);
143
```

JTransformer - Src for id 52465

```
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;
}
```

OK

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)

Prolog Console

```
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 ?-
```

Query factbase or run CTs

PEF Navigator

```
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, [], type
    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,
```

Screenshot and explanation of behaviour TO BE UPDATED

Conclusions

- Logic Programming for Java Software Analysis
- Java program elements → Facts
- Java program analyses → Predicates
- Complete development & runtime environment

Learn More

- Go to the JTransformer website
- Follow the „Getting Started“ Tutorial
- Explore the website
- Try out
- Register in the mailing list
- Ask